Java Platform, Standard Edition

Security Developer's Guide

Release 27
G53789-01
September 2026

Java Platform, Standard Edition Security Developer’s Guide, Release 27

G53789-01

Copyright © 1993, 2026, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documents i
Conventions i

1 General Security

Terms and Definitions 1
Java Security Overview 4
- Introduction to Java Security 4
- Java Language Security and Bytecode Verification 5
- Basic Security Architecture 6
  - Security Providers 6
- Java Cryptography 8
- Public Key Infrastructure 9
  - Key and Certificate Storage 9
  - Public Key Infrastructure Tools 10
- Authentication 11
- Secure Communication 12
  - TLS and DTLS Protocols 13
  - Simple Authentication and Security Layer (SASL) 13
  - Generic Security Service API and Kerberos 14
- XML Signature 14
- Java API for XML Processing (JAXP) 15
- Security Tools Summary 15
  - The JAR Signing and Verification Tool 16
  - The Key and Certificate Management Tool 16
  - Restricting the Java Platform from Reading Passwords from the Standard Input Stream 17
- Built-In Providers 17

The Security Properties File 17
- Specifying an Alternative Security Properties File 18

Copyright © 1993, 2026, Oracle and/or its affiliates.

Statically Setting a Security Property in a Security Properties File 18
Dynamically Setting a Security Property in Application Code 19
Troubleshooting Security Properties 19
Including a Security Properties File 19
Order of include Directives in Security Properties Files 21
Security Considerations and include Directives in Security Properties Files 21
Examples of Using include Directives 22
Standard Algorithm Names 23
The Security Manager Is Permanently Disabled 23
Troubleshooting Security 31
The java.security.debug System Property 31
Debug Statement Output Format 34
The java -XshowSettings:security Option 34
Enabling Debugging in Java Generic Security Services 35

2 Java Cryptography Architecture (JCA) Reference Guide

Introduction to Java Cryptography Architecture 1
JCA Design Principles 2
Provider Architecture 3
Cryptographic Service Providers 3
How Providers Are Actually Implemented 5
Keystores 6
Engine Classes and Algorithms 7
Core Classes and Interfaces 8
The Provider Class 9
How Provider Implementations Are Requested and Supplied 10
Installing Providers 12
Provider Class Methods 12
The Security Class 12
Managing Providers 12
Security Properties 13
The SecureRandom Class 14
Creating a SecureRandom Object 14
Seeding or Re-Seeding the SecureRandom Object 15
Using a SecureRandom Object 15
Generating Seed Bytes 15
The MessageDigest Class 15
Creating a MessageDigest Object 16
Updating a Message Digest Object 16
Computing the Digest 16
The Signature Class 17

Copyright © 1993, 2026, Oracle and/or its affiliates.

Signature Object States 17
Creating a Signature Object 18
Initializing a Signature Object 18
Signing with a Signature Object 18
Verifying with a Signature Object 19
The SignedObject Class 20
The Cipher Class 21
Other Cipher-based Classes 28
The Cipher Stream Classes 29
The SealedObject Class 31
The KEM Class 33
The Mac Class 33
Key Interfaces 35
The KeyPair Class 37
Key Specification Interfaces and Classes 37
The KeySpec Interface 38
The KeySpec Subinterfaces 38
The EncodedKeySpec Class 38
Generators and Factories 39
The KeyFactory Class 39
The SecretKeyFactory Class 41
The KeyPairGenerator Class 42
The KeyGenerator Class 44
The KDF Class 45
The KeyAgreement Class 47
Key Management 49
The KeyStore Class 51
The BinaryEncodable Interface 54
The Privacy-Enhanced Mail (PEM) Transport Format 56
The PEMEncoder Class 56
The PEMDecoder Class 57
The EncryptedPrivateKeyInfo Class 58
The PEM Class 60
Algorithm Parameters Classes 61
The AlgorithmParameterSpec Interface 61
The AlgorithmParameters Class 62
The AlgorithmParameterGenerator Class 63
The CertificateFactory Class 64
Standard Names 65
How the JCA Might Be Used in a SSL/TLS Implementation 66
Cryptographic Strength Configuration 68
Jurisdiction Policy File Format 71

Copyright © 1993, 2026, Oracle and/or its affiliates.

How to Make Applications Exempt from Cryptographic Restrictions 73
Packaging Your Application 77
Additional JCA Code Samples 77
Computing a MessageDigest Object 77
Generating a Pair of Keys 79
Generating and Verifying a Signature Using Generated Keys 80
Generating/Verifying Signatures Using Key Specifications and KeyFactory 81
Generating Random Numbers 83
Determining If Two Keys Are Equal 84
Reading Base64-Encoded Certificates 84
Parsing a Certificate Reply 85
Using Encryption 85
Using Password-Based Encryption 86
Encapsulating and Decapsulating Keys 87
Sample Programs for Diffie-Hellman Key Exchange, AES/GCM, HMAC-SHA256, and HPKE 88
Diffie-Hellman Key Exchange between Two Parties 88
Diffie-Hellman Key Exchange between Three Parties 92
AES/GCM Example 94
HMAC-SHA256 Example 94
Securely Exchanging Messages with HPKE and the HPKEParameterSpec Class 95

3 How to Implement a Provider in the Java Cryptography Architecture

Who Should Read This Document 1
Notes on Terminology 1
Introduction to Implementing Providers 1
Engine Classes and Corresponding Service Provider Interface Classes 2
Steps to Implement and Integrate a Provider 5
Step 1: Write your Service Implementation Code 5
Step 1.1: Consider Additional JCA Provider Requirements and Recommendations for Encryption Implementations 6
Step 2: Give your Provider a Name 7
Step 3: Write Your Master Class, a Subclass of Provider 7
Step 3.1: Create a Provider That Uses String Objects to Register Its Services 7
Step 3.2: Create a Provider That Uses Provider.Service 10
Step 3.3: Specify Additional Information for Cipher Implementations 12
Step 4: Create a Module Declaration for Your Provider 14
Step 5: Compile Your Code 15
Step 6: Place Your Provider in a JAR File 15
Step 7: Sign Your JAR File, If Necessary 15
Step 7.1: Get a Code-Signing Certificate 16
Step 7.2: Sign Your Provider 18

Copyright © 1993, 2026, Oracle and/or its affiliates.

Step 8: Prepare for Testing 18
Step 8.1: Configure the Provider 18
Step 9: Write and Compile Your Test Programs 20
Step 10: Run Your Test Programs 21
Step 11: Apply for U.S. Government Export Approval If Required 22
Step 12: Document Your Provider and Its Supported Services 23
Step 12.1: Indicate Whether Your Implementation is Cloneable for Message Digests and MACs 23
Step 13: Make Your Class Files and Documentation Available to Clients 25
Further Implementation Details and Requirements 25
Alias Names 25
Service Interdependencies 26
Default Initialization 28
Default Key Pair Generator Parameter Requirements 28
The Provider.Service Class 29
Signature Formats 30
DSA Interfaces and their Required Implementations 30
RSA Interfaces and their Required Implementations 33
Diffie-Hellman Interfaces and their Required Implementations 34
Interfaces for Other Algorithm Types 36
Algorithm Parameter Specification Interfaces and Classes 36
Key Specification Interfaces and Classes Required by Key Factories 39
Secret-Key Generation 44
Adding New Object Identifiers 44
Ensuring Exportability 46
Sample Code for MyProvider 46

4 JDK Providers Documentation

Introduction to JDK Providers 1
Import Limits on Cryptographic Algorithms 2
Cipher Transformations 3
SecureRandom Implementations 3
The SunPKCS11 Provider 4
The SUN Provider 4
The SunRsaSign Provider 9
The SunJSSE Provider 10
The SunJCE Provider 18
The SunJGSS Provider 25
The SunSASL Provider 25
The XMLDSig Provider 26
The SunPCSC Provider 26

Copyright © 1993, 2026, Oracle and/or its affiliates.

The SunMSCAPI Provider 27
The SunEC Provider 28
The Apple Provider 31
The JdkLDAP Provider 31
The JdkSASL Provider 32

5 PKCS#11 Reference Guide

SunPKCS11 Provider 1
SunPKCS11 Requirements 2
SunPKCS11 Configuration 2
Accessing Network Security Services (NSS) 8
Troubleshooting PKCS#11 10
Disabling PKCS#11 Providers and/or Individual PKCS#11 Mechanisms 11
Application Developers 12
- Token Login 12
- Token Keys 13
- Delayed Provider Selection 14
- JAAS KeyStoreLoginModule 15
- Tokens as JSSE Keystore and Trust Stores 16

Using keytool and jarsigner with PKCS#11 Tokens 16
Provider Developers 18
- Provider Services 18
- Parameter Support 19

SunPKCS11 Provider Supported Algorithms 20
SunPKCS11 Provider KeyStore Requirements 26
Example Provider 28

6 Java Authentication and Authorization Service (JAAS)

Java Authentication and Authorization Service (JAAS) Reference Guide 1
- Who Should Read This Document 2
- Related Documentation 2
- Core Classes and Interfaces 2
  - Common Classes 2
  - Authentication Classes and Interfaces 8
- JAAS Authorization 10
- Appendix A: JAAS Settings in the java.security Security Properties File 11
  - Login Configuration Provider 12
  - Login Configuration URLs 13
- Appendix B: JAAS Login Configuration File 13
  - Login Configuration File Structure and Contents 13

Copyright © 1993, 2026, Oracle and/or its affiliates.

Where to Specify Which Login Configuration File Should Be Used 15
JAAS Authentication Tutorial 16
The Authentication Tutorial Code 17
SampleAcn.java 17
SampleLoginModule.java and SamplePrincipal.java 27
The Login Configuration 34
The Login Configuration File for the JAAS Authentication Tutorial 35
Running the JAAS Authentication Tutorial Code 35
Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide 37
Introduction to LoginModule 38
Steps to Implement a LoginModule 39
Step 1: Understand the Authentication Technology 39
Step 2: Name the LoginModule Implementation 40
Step 3: Implement the LoginModule Interface 40
Step 4: Choose or Write a Sample Application 44
Step 5: Compile the LoginModule and Application 45
Step 6: Prepare for Testing 45
Step 7: Test Use of the LoginModule 45
Step 8: Document Your LoginModule Implementation 46
Step 9: Make LoginModule JAR File and Documents Available 46
Migrating from the Deprecated for Removal Subject.getSubject and Subject.doAs Methods to the Subject.current and Subject.callAs Methods 46

7 Java Generic Security Services (Java GSS-API)

Introduction to JAAS and Java GSS-API Tutorials 1
When to Use Java GSS-API Versus JSSE 2
Use of Java GSS-API for Secure Message Exchanges Without JAAS Programming 3
Overview of the Client and Server Applications 4
The SampleClient and SampleServer Code 4
Kerberos User and Service Principal Names 17
The Login Configuration File 18
The useSubjectCredsOnly System Property 19
Running the SampleClient and SampleServer Programs 20
JAAS Authentication 23
The Authentication Tutorial Code 23
The Login Configuration 25
Running the Code 25
Use of JAAS Login Utility 27
What You Need to Know About the Login Utility 27
Application and Other File Requirements 28
The Sample Application Program 28

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Login Configuration File 28
Running the Sample Program with the Login Utility 29
Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges 30
Before You Start: Recommended Reading 31
Overview of the Client and Server Applications 31
Kerberos User and Service Principal Names 32
The Login Configuration File 33
Running the SampleClient and SampleServer Programs 35
Using Credentials Delegated from the Client 38
Kerberos Requirements 39
Setting Properties to Indicate the Default Realm and KDC 40
Locating the krb5.conf Configuration File 40
Naming Conventions for Realm Names and Hostnames 41
Cross-Realm Authentication 41
Source Code for JAAS and Java GSS-API Tutorials 42
Accessing Native GSS-API 53
Advanced Security Programming in Java SE Authentication, Secure Communication and Single Sign-On 57
Part I : Secure Authentication using the Java Authentication and Authorization Service (JAAS) 58
Exercise 1: Using the JAAS API 58
Exercise 2: Configuring JAAS for Kerberos Authentication 59
Part II : Secure Communications using the Java SE Security API 60
Exercise 3: Using the Java Generic Security Service (GSS) API 61
Exercise 4: Using the Java SASL API 63
Part III : Deploying for Single Sign-On in a Kerberos Environment 65
Exercise 5: Deploying for Single Sign-On 65
Part IV : Secure Communications Using Stronger Encryption Algorithms 66
Exercise 6: Configuring to Use Stronger Encryption Algorithms in a Kerberos Environment, to Secure the Communication 66
Part V : Secure Authentication Using SPNEGO Java GSS Mechanism 67
Exercise 7: Using the Java Generic Security Services (GSS) API with SPNEGO 67
Part VI: HTTP/SPNEGO Authentication 69
Exercise 8: Using HTTP/SPNEGO Authentication 69
Source Code for Advanced Security Programming in Java SE Authentication, Secure Communication and Single Sign-On 74
Appendix A: Setting up Kerberos Accounts 103
The Kerberos 5 GSS-API Mechanism 104
Kerberos System Properties, Security Properties, and Environment Variables 106
Troubleshooting JGSS 113
Troubleshooting Logins 113

Copyright © 1993, 2026, Oracle and/or its affiliates.

8 Java Secure Socket Extension (JSSE) Reference Guide

Introduction to JSSE 1
- JSSE Features and Benefits 2
- JSSE Standard API 2
- SunJSSE Provider 3
- JSSE Related Documentation 3

JSSE Classes and Interfaces 4
- JSSE Core Classes and Interfaces 5
- SocketFactory and ServerSocketFactory Classes 5
- SSLSocketFactory and SSLServerSocketFactory Classes 5
  - Obtaining an SSLSocketFactory 6
- SSLSocket and SSLServerSocket Classes 6
  - Obtaining an SSLSocket 6
  - Cipher Suite Choice and Remote Entity Verification 7
- SSLEngine Class 7
  - SSLEngine Methods 9
  - Understanding SSLEngine Operation Statuses 10
  - SSLEngine for TLS Protocols 15
  - SSLEngine for DTLS Protocols 21
  - Dealing With Blocking Tasks 30
  - Shutting Down a TLS/DTLS Connection 30
- SSLSession and ExtendedSSLSession 31
  - TLS Keying Material Exporters 32
- HttpsURLConnection Class 33
  - Setting the Assigned SSLSocketFactory 33
  - Setting the Assigned HostnameVerifier 33
- Support Classes and Interfaces 34
  - SSLContext Class 34
  - TrustManager Interface 37
  - TrustManagerFactory Class 37
  - X509TrustManager Interface 40
  - X509ExtendedTrustManager Class 43
  - KeyManager Interface 46
  - KeyManagerFactory Class 46
  - X509KeyManager Interface 47
  - X509ExtendedKeyManager Class 48
  - Relationship Between a TrustManager and a KeyManager 49
- Secondary Support Classes and Interfaces 49
  - SSLParameters Class 49
  - SSLSessionContext Interface 50
  - SSLSessionBindingListener Interface 50

Copyright © 1993, 2026, Oracle and/or its affiliates.

SSLSessionBindingEvent Class 50
HandShakeCompletedListener Interface 51
HandShakeCompletedEvent Class 51
HostnameVerifier Interface 51
X509Certificate Class 51
AlgorithmConstraints Interface 52
StandardConstants Class 52
SNIServerName Class 52
SNIMatcher Class 52
SNIHostName Class 53
Customizing JSSE 53
How to Specify a java.lang.System Property 65
How to Specify a java.security.Security Property 65
Customizing the X509Certificate Implementation 66
Specifying Default Enabled Cipher Suites 67
Specifying an Alternative HTTPS Protocol Implementation 68
Customizing the Provider Implementation 68
Registering the Cryptographic Provider Statically 68
Registering the Cryptographic Service Provider Dynamically 69
Provider Configuration 69
Configuring the Preferred Provider for Specific Algorithms 69
Customizing the Default Keystores and Truststores, Store Types, and Store Passwords 70
Customizing the Default Key Managers and Trust Managers 72
Disabled and Restricted Cryptographic Algorithms 73
Legacy Cryptographic Algorithms 74
Customizing the Encryption Algorithm Providers 75
Customizing the Size of Ephemeral Diffie-Hellman Keys 75
Customizing the Maximum Fragment Length Negotiation (MFLN) Extension 76
Configuring the Maximum and Minimum Packet Size 77
Limiting Amount of Data Algorithms May Encrypt with a Set of Keys 77
Resuming Session Without Server-Side State 78
Specifying That close_notify Alert Is Sent When One Is Received 79
Enabling certificate_authorities Extension for Server Certificate Selection 79
SunJSSE Renegotiation Interoperability Modes 79
Workarounds and Alternatives to SSL/TLS Renegotiation 81
Allow Unsafe Server Certificate Change in SSL/TLS Renegotiations 82
Client-Driven OCSP and OCSP Stapling 82
Client-Driven OCSP and Certificate Revocation 83
OCSP Stapling and Certificate Revocation 84
OCSP Stapling Configuration Properties 85
Configuring Default Extensions 87
Configuring TLS 1.3 Certificate Compression 88

Copyright © 1993, 2026, Oracle and/or its affiliates.

Specifying Signature Schemes That Can Be Used over the TLS/DTLS Protocols 88
Customizing the Supported Named Groups for TLS/DTLS Key Exchange 88
Enabling Hybrid Key Exchange Schemes 88
Setting the Maximum Certificate Chain Length Accepted from the Client or Server During TLS/DTLS Handshaking 89
Support for the TLS Session Hash and Extended Master Secret Extension 90
Hardware Acceleration and Smartcard Support 90
Configuring JSSE to Use Smartcards as Keystores and Truststores 91
Multiple and Dynamic Keystores 91
The PKCS12 Keystore Format 92
Server Name Indication (SNI) Extension 92
TLS Application Layer Protocol Negotiation 94
Setting up ALPN on the Client 95
Setting up Default ALPN on the Server 96
Setting up Custom ALPN on the Server 97
Determining Negotiated ALPN Value during Handshaking 99
Reading and Writing ALPN Values with the SunJSSE Provider 102
ALPN Related Classes and Methods 104
Troubleshooting JSSE 105
Configuration Problems 105
SSLHandshakeException: No Available Authentication Scheme, Handshake Failure 105
CertificateException While Handshaking 106
Runtime Exception: SSL Service Not Available 106
Runtime Exception: "No available certificate corresponding to the SSL cipher suites which are enabled" 107
Runtime Exception: No Cipher Suites in Common 107
Socket Disconnected After Sending ClientHello Message 108
SunJSSE Cannot Find a JCA Provider That Supports a Required Algorithm and Causes a NoSuchAlgorithmException 109
Exception Thrown When Obtaining Application Resources from a Virtual Host Web Server that Requires an SNI Extension 109
IllegalArgumentException When RC4 Cipher Suites are Configured for DTLS 110
Debugging Utilities 110
Debugging TLS Connections 112
Compatibility Risks and Known Issues 132
Code Examples 132
Converting an Unsecure Socket to a Secure Socket 132
Running the JSSE Sample Code 135
Creating a Keystore to Use with JSSE 142
Using the Server Name Indication (SNI) Extension 146
Typical Client-Side Usage Examples 146
Typical Server-Side Usage Examples 147
Working with Virtual Infrastructures 148

Copyright © 1993, 2026, Oracle and/or its affiliates.

Provider Pluggability 152
Transport Layer Security (TLS) Protocol Overview 152
How TLS Works 153
Cryptographic Processes 153
Secret-Key Cryptography 153
Public-Key Cryptography 154
Comparison Between Secret-Key and Public-Key Cryptography 154
Public Key Certificates 155
Cryptographic Hash Functions 155
Message Authentication Code 156
Digital Signatures 156
The TLS 1.3 Handshake 156
The TLS 1.3 Protocol 156
Session Resumption with a Pre-Shared Key 161
Post-Handshake Messages 164
Compatibility Risks and Known Issues 165
The TLS 1.2 Handshake 165
The TLS 1.2 Protocol 166
Datagram Transport Layer Security (DTLS) Protocol 168
The DTLS Handshake 169

9 Java PKI Programmer's Guide

PKI Programmer's Guide Overview 1
Introduction to Public Key Certificates 2
X.509 Certificates and Certificate Revocation Lists (CRLs) 3
Core Classes and Interfaces 6
Basic Certification Path Classes 7
The CertPath Class 7
The CertificateFactory Class 9
The CertPathParameters Interface 10
Certification Path Validation Classes 11
The CertPathValidator Class 11
The CertPathValidatorResult Interface 12
Certification Path Building Classes 13
The CertPathBuilder Class 13
The CertPathBuilderResult Interface 14
Certificate/CRL Storage Classes 14
The CertStore Class 15
The CertStoreParameters Interface 16
The CertSelector and CRLSelector Interfaces 17
PKIX Classes 22

Copyright © 1993, 2026, Oracle and/or its affiliates.

The TrustAnchor Class 23
The PKIXParameters Class 24
The PKIXCertPathValidatorResult Class 26
The PolicyNode Interface and PolicyQualifierInfo Class 26
The PKIXBuilderParameters Class 28
The PKIXCertPathBuilderResult Class 29
The PKIXCertPathChecker Class 30
Using PKIXCertPathChecker in Certificate Path Validation 35
Implementing a Service Provider 40
Steps to Implement and Integrate a Provider 40
Service Interdependencies 42
Certification Path Parameter Specification Interfaces 42
Certification Path Result Specification Interfaces 43
Certification Path Exception Classes 43
Appendix A: Standard Names 44
Appendix B: The CertPath Implementation in the SUN Provider 44
Appendix C: OCSP Support 47
Enable OSCP Nonce Extension 49
Maximum Allowable Clock Skew 50
Fallback Option for POST-Only OCSP Requests 50
Appendix D: CertPath Implementation in JdkLDAP Provider 50
Appendix E: Disabling Cryptographic Algorithms 51

10 Java SASL API Programming and Deployment Guide

Java SASL API Overview 2
Creating the Mechanisms 2
Passing Input to the Mechanisms 3
Using the Mechanisms 3
Using the Negotiated Security Layer 5
How SASL Mechanisms are Installed and Selected 6
The SunSASL Provider 7
The SunSASL Provider Client Mechanisms 7
The SunSASL Provider Server Mechanisms 12
The JdkSASL Provider 13
The JdkSASL Provider Client Mechanism 13
The JdkSASL Provider Server Mechanism 15
Debugging and Monitoring 15
Implementing a SASL Security Provider 16

Copyright © 1993, 2026, Oracle and/or its affiliates.

11 XML Digital Signature API Overview and Tutorial

Package Hierarchy 1
Service Providers 2
Introduction to XML Signatures 3
Example of an XML Signature 3
XML Signature Secure Validation Mode 4
XML Digital Signature API Examples 5
Validate Example 5
Validating an XML Signature 9
Instantiating the Document that Contains the Signature 9
Specifying the Signature Element to be Validated 9
Creating a Validation Context 10
Unmarshalling the XML Signature 10
Validating the XML Signature 10
Using KeySelectors 11
GenEnveloped Example 12
Generating an XML Signature 15
Instantiating the Document to be Signed 16
Creating a Public Key Pair 16
Creating a Signing Context 16
Assembling the XML Signature 17
Generating the XML Signature 18
Printing or Displaying the Resulting Document 18

12 Java API for XML Processing (JAXP) Security Guide

Security Issues in XML Processing 1
XML External Entity Injection Attack 1
External Resources Supported by XML, Schema, and XSLT Standards 1
Exponential Entity Expansion Attack 3
Configuring JAXP for Secure XML Processing 3
Secure XML Processing Best Practices 4
JAXP APIs 5
Factories and Processors 5
Configuring with JAXP Properties 5
Setting JAXP Properties Through JAXP Factories 6
Setting JAXP Properties as System Properties 7
Setting JAXP Properties in a JAXP Configuration File 8
Scope and Order of Setting JAXP Properties 10
Security-Related Properties 10
API-Defined Properties 11

Copyright © 1993, 2026, Oracle and/or its affiliates.

JDK-Specific Properties 13
Composite Processors 21
Using Resolvers and Catalogs 22
Java XML Resolvers 22
The Catalog API 24
The Built-In Catalog 24
Handling Errors from JAXP Properties 25
General Recommendations for JAXP Security 27
Appendix A: Glossary of Java API for XML Processing Terms and Definitions 27
Appendix B: Java and JDK XML Features and Properties Naming Convention 28

Copyright © 1993, 2026, Oracle and/or its affiliates.

Preface

This guide provides information about the Java security technology, tools, and implementations of commonly used security algorithms, mechanisms, and protocols on the Java Platform, Standard Edition (Java SE).

Audience

This document is intended for experienced developers who build applications using the comprehensive Java security framework. It is also intended for the user or administrator with a set of tools to securely manage applications.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

- Serialization Filtering in *Java Platform, Standard Edition Core Libraries*
- RMI Security Recommendations in *Java Platform, Standard Edition Java Remote Method Invocation User's Guide*

Conventions

The following text conventions are used in this document:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Copyright © 1993, 2026, Oracle and/or its affiliates.

1

General Security

Terms and Definitions list commonly used cryptography terms and their definitions.

Java Security Overview provides an overview of the motivation of major security features, an introduction to security classes and their usage, a discussion of the impact of the security architecture on code, and thoughts on writing security-sensitive code.

Certain aspects of Java security may be customized by setting security properties in the securities properties file. The Security Properties File describes this file.

Java Security Standard Algorithm Names Specification describes the set of standard names for algorithms, certificate and keystore types that Java SE requires and uses.

Troubleshooting Security lists options for the `java.security.debug` system property that enable you to monitor security access.

Terms and Definitions

The following are commonly used cryptography terms and their definitions.

authentication
The process of confirming the identity of a party with whom one is communicating.

certificate
A digitally signed statement vouching for the identity and public key of an entity (person, company, and so on). Certificates can either be self-signed or issued by a Certificate Authority (CA), an entity that is trusted to issue valid certificates for other entities. Well-known CAs include Comodo, DigiCert, and GoDaddy. X.509 is a common certificate format that can be managed by the JDK's `keytool`.

cipher suite
A combination of cryptographic parameters that define the security algorithms and key sizes used for authentication, key agreement, encryption, and integrity protection.

cryptographic hash function
An algorithm that is used to produce a relatively small fixed-size string of bits (called a hash) from an arbitrary block of data. A cryptographic hash function is similar to a checksum and has three primary characteristics: it’s a one-way function, meaning that it is not possible to produce the original data from the hash; a small change in the original data produces a large change in the resulting hash; and it doesn’t require a cryptographic key.

Cryptographic Service Provider (CSP)
Sometimes referred to simply as providers for short, the Java Cryptography Architecture (JCA) defines it as a package (or set of packages) that implements one or more engine classes for specific cryptographic algorithms. An engine class defines a cryptographic service in an abstract fashion without a concrete implementation.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Datagram Transport Layer Security (DTLS) Protocol
A protocol that manages client and server authentication, data integrity, and encrypted communication between the client and server based on an unreliable transport channel such as UDP.

decryption
See encryption/decryption.

digital signature
A digital equivalent of a handwritten signature. It is used to ensure that data transmitted over a network was sent by whoever claims to have sent it and that the data has not been modified in transit. For example, an RSA-based digital signature is calculated by first computing a cryptographic hash of the data and then encrypting the hash with the sender's private key.

encryption/decryption
Encryption is the process of using a complex algorithm to convert an original message (cleartext) to an encoded message (ciphertext) that is unintelligible unless it is decrypted. Decryption is the inverse process of producing cleartext from ciphertext.
The algorithms used to encrypt and decrypt data typically come in two categories: secret key (symmetric) cryptography and public key (asymmetric) cryptography.

endpoint identification
An IPv4 or IPv6 address used to identify an endpoint on the network.
Endpoint identification procedures are handled during SSL/TLS handshake.

handshake protocol
The negotiation phase during which the two socket peers agree to use a new or existing session. The handshake protocol is a series of messages exchanged over the record protocol. At the end of the handshake, new connection-specific encryption and integrity protection keys are generated based on the key agreement secrets in the session.

Hybrid Public Key Encryption (HPKE)
A public key encryption scheme for encrypting arbitrary-sized plaintexts with a recipient's public key. It combines a key encapsulation mechanism (KEM), a key derivation function (KDF), and an authenticated encryption with additional data (AEAD) cipher.

java-home
Variable placeholder used throughout this document to refer to the directory where the Java Development Kit (JDK) is installed.

key agreement
A method by which two parties cooperate to establish a common key. Each side generates some data, which is exchanged. These two pieces of data are then combined to generate a key. Only those holding the proper private initialization data can obtain the final key. Diffie-Hellman (DH) is the most common example of a key agreement algorithm.

Key Encapsulation Mechanism (KEM)
An encryption technique for securing symmetric keys using public key cryptography. In the encapsulation process, the sender reads in the receiver's public key and generates a secret key and a key encapsulation message. The key encapsulation message is sent to the receiver. In the decapsulation process, the receiver uses its own private key to recover the same secret key from the key encapsulation message.

key exchange
A method by which keys are exchanged. One side generates a private key and encrypts it using the peer's public key (typically RSA). The data is transmitted to the peer, who decrypts the key using the corresponding private key.

Copyright © 1993, 2026, Oracle and/or its affiliates.

key manager/trust manager
Key managers and trust managers use keystores for their key material. A key manager manages a keystore and supplies public keys to others as needed (for example, for use in authenticating the user to others). A trust manager decides who to trust based on information in the truststore it manages.

Keyed-Hash Message Code (HMAC)
A specific type of message authentication code that involves a cryptographic hash function and a secret cryptographic key.

Keyed-Hash Message Code (HMAC)-based Extract-and-Expand Key Derivation Function (HKDF)
A function used for key generation and validation.

keystore/truststore
A keystore is a database of key material. Key material is used for a variety of purposes, including authentication and data integrity. Various types of keystores are available, including PKCS12 and the macOS KeychainStore.
Generally speaking, keystore information can be grouped into two categories: key entries and trusted certificate entries. A key entry consists of an entity's identity and its private key, and can be used for a variety of cryptographic purposes. In contrast, a trusted certificate entry contains only a public key in addition to the entity's identity. Thus, a trusted certificate entry can't be used where a private key is required, such as in a `javax.net.ssl.KeyManager`. In the JDK implementation of PKCS12, a keystore may contain both key entries and trusted certificate entries.
A truststore is a keystore that is used when making decisions about what to trust. If you receive data from an entity that you already trust, and if you can verify that the entity is the one that it claims to be, then you can assume that the data really came from that entity.
An entry should only be added to a truststore if the user trusts that entity. By either generating a key pair or by importing a certificate, the user gives trust to that entry. Any entry in the truststore is considered a trusted entry.
It may be useful to have two different keystore files: one containing just your key entries, and the other containing your trusted certificate entries, including CA certificates. The former contains private information, whereas the latter does not. Using two files instead of a single keystore file provides a cleaner separation of the logical distinction between your own certificates (and corresponding private keys) and others' certificates. To provide more protection for your private keys, store them in a keystore with restricted access, and provide the trusted certificates in a more publicly accessible keystore if needed.

message authentication code (MAC)
Provides a way to check the integrity of information transmitted over or stored in an unreliable medium, based on a secret key. Typically, MACs are used between two parties that share a secret key in order to validate information transmitted between these parties.
A MAC mechanism that is based on cryptographic hash functions is referred to as HMAC.

public-key cryptography
A cryptographic system that uses an encryption algorithm in which two keys are produced. One key is made public, whereas the other is kept private. The public key and the private key are cryptographic inverses; what one key encrypts only the other key can decrypt. Public-key cryptography is also called asymmetric cryptography.

Record Protocol
A protocol that packages all data (whether application-level or as part of the handshake process) into discrete records of data much like a TCP stream socket converts an application byte stream into network packets. The individual records are then protected by the current encryption and integrity protection keys.

Copyright © 1993, 2026, Oracle and/or its affiliates.

secret-key cryptography
A cryptographic system that uses an encryption algorithm in which the same key is used both to encrypt and decrypt the data. Secret-key cryptography is also called symmetric cryptography.

Secure Sockets Layer (SSL) Protocol
A protocol that manages client and server authentication, data integrity, and encrypted communication between the client and server. SSL has been renamed to Transport Layer Security (TLS).

session
A named collection of state information including authenticated peer identity, cipher suite, and key agreement secrets that are negotiated through a secure socket handshake and that can be shared among multiple secure socket instances.

Transport Layer Security (TLS) Protocol
A protocol that manages client and server authentication, data integrity, and encrypted communication between the client and server based on a reliable transport channel such as TCP.

trust manager
See key manager/trust manager.

truststore
See keystore/truststore.

Java Security Overview

Java security includes a large set of APIs, tools, and implementations of commonly-used security algorithms, mechanisms, and protocols. The Java security APIs span a wide range of areas, including cryptography, public key infrastructure, secure communication, authentication, and access control. Java security technology provides the developer with a comprehensive security framework for writing applications, and also provides the user or administrator with a set of tools to securely manage applications.

Introduction to Java Security

The JDK is designed with a strong emphasis on security. At its core, the Java language itself is type-safe and provides automatic garbage collection, enhancing the robustness of application code. A secure class loading and verification mechanism ensures that only legitimate Java code is executed. The Java security architecture includes a large set of application programming interfaces (APIs), tools, and implementations of commonly-used security algorithms, mechanisms, and protocols.

The Java security APIs span a wide range of areas. Cryptographic and public key infrastructure (PKI) interfaces provide the underlying basis for developing secure applications.

The APIs allow for multiple interoperable implementations of algorithms and other security services. Services are implemented in *providers*, which are plugged into the JDK through a standard interface that makes it easy for applications to obtain security services without having to know anything about their implementations. This allows developers to focus on how to integrate security into their applications, rather than on how to actually implement complex security mechanisms.

The JDK includes a number of providers that implement a core set of security services. It also allows for additional custom providers to be installed. This enables developers to extend the platform with new security mechanisms.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The JDK is divided into modules. Modules that contain security APIs include the following:

Table 1-1 Modules That Contain Security APIs

Module	Description
`java.base`	Defines the foundational APIs of Java SE. Contained packages include `java.security`, `javax.crypto`, `javax.net.ssl`, and `javax.security.auth`.
`java.security.jgss`	Defines the Java binding of the IETF Generic Security Services API (GSS-API). This module also contains GSS-API mechanisms including Kerberos v5 and SPNEGO.
`java.security.sasl`	Defines Java support for the IETF Simple Authentication and Security Layer (SASL). This module also contains SASL mechanisms including DIGEST-MD5, CRAM-MD5, and NTLM,
`java.smartcardio`	Defines the Java Smart Card I/O API.
`java.xml.crypto`	Defines the API for XML cryptography.
`jdk.jartool`	Defines APIs for signing JAR files.
`jdk.security.auth`	Provides implementations of the `javax.security.auth.*` interfaces and various authentication modules.
`jdk.security.jgss`	Defines Java extensions to the GSS-API and an implementation of the SASL GSS-API mechanism.

Java Language Security and Bytecode Verification

The Java language is designed to be type-safe and easy to use. It provides automatic memory management, garbage collection, and range-checking on arrays. This reduces the overall programming burden placed on developers, leading to fewer subtle programming errors and to safer, more robust code.

A compiler translates Java programs into a machine-independent bytecode representation. A bytecode verifier is invoked to ensure that only legitimate bytecodes are executed in the Java runtime. It checks that the bytecodes conform to the Java Language Specification and do not violate Java language rules or namespace restrictions. The verifier also checks for memory management violations, stack underflows or overflows, and illegal data typecasts. Once bytecodes have been verified, the Java runtime prepares them for execution.

In addition, the Java language defines different access modifiers that can be assigned to Java classes, methods, and fields, enabling developers to restrict access to their class implementations as appropriate. The language defines four distinct access levels:

- `private`: Most restrictive modifier; access is not allowed outside the particular class in which the private member (a method, for example) is defined.
- `protected`: Allows access to any subclass or to other classes within the same package.
- Package-private: If not specified, then this is the default access level; allows access to classes within the same package.
- `public`: No longer guarantees that the element is accessible everywhere; accessibility depends upon whether the package containing that element is exported by its defining module and whether that module is readable by the module containing the code that is attempting to access it.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Basic Security Architecture

The JDK defines a set of APIs spanning major security areas, including cryptography, public key infrastructure, authentication, and secure communication. The APIs allow developers to easily integrate security into their application code.

The APIs are designed around the following principles:

Implementation independence
Applications do not need to implement security themselves. Rather, they can request security services from the JDK. Security services are implemented in providers (see the section Security Providers), which are plugged into the JDK via a standard interface. An application may rely on multiple independent providers for security functionality.

Implementation interoperability
Providers are interoperable across applications. Specifically, an application is not bound to a specific provider if it does not rely on default values from the provider.

Algorithm extensibility
The JDK includes a number of built-in providers that implement a basic set of security services that are widely used today. However, some applications may rely on emerging standards not yet implemented, or on proprietary services. The JDK supports the installation of custom providers that implement such services.

Security Providers

The `java.security.Provider` class encapsulates the notion of a security provider in the Java platform. It specifies the provider's name and lists the security services it implements. Multiple providers may be configured at the same time and are listed in order of preference. When a security service is requested, the highest priority provider that implements that service is selected.

Applications rely on the relevant `getInstance` method to request a security service from an underlying provider.

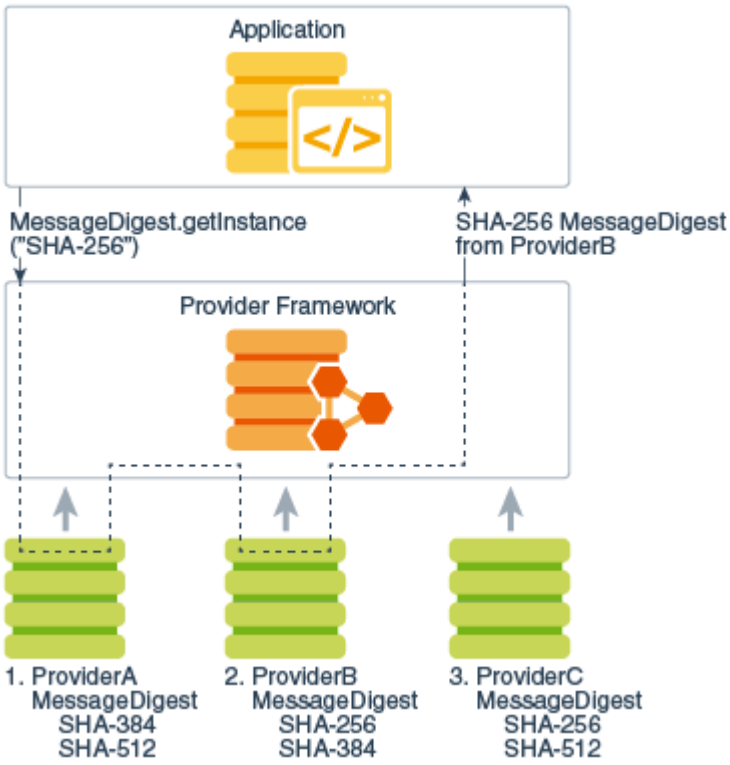
For example, message digest creation represents one type of service available from providers. To request an implementation of a specific message digest algorithm, call the method `java.security.MessageDigest.getInstance`. The following statement requests a SHA-256 message digest implementation without specifying a provider name:

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
```

The following figure illustrates how this statement obtains a SHA-256 message digest implementation. The providers are searched in preference order, and the implementation from the first provider supplying that particular algorithm, `ProviderB`, is returned.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 1-1 Request SHA-256 Message Digest Implementation Without Specifying Provider



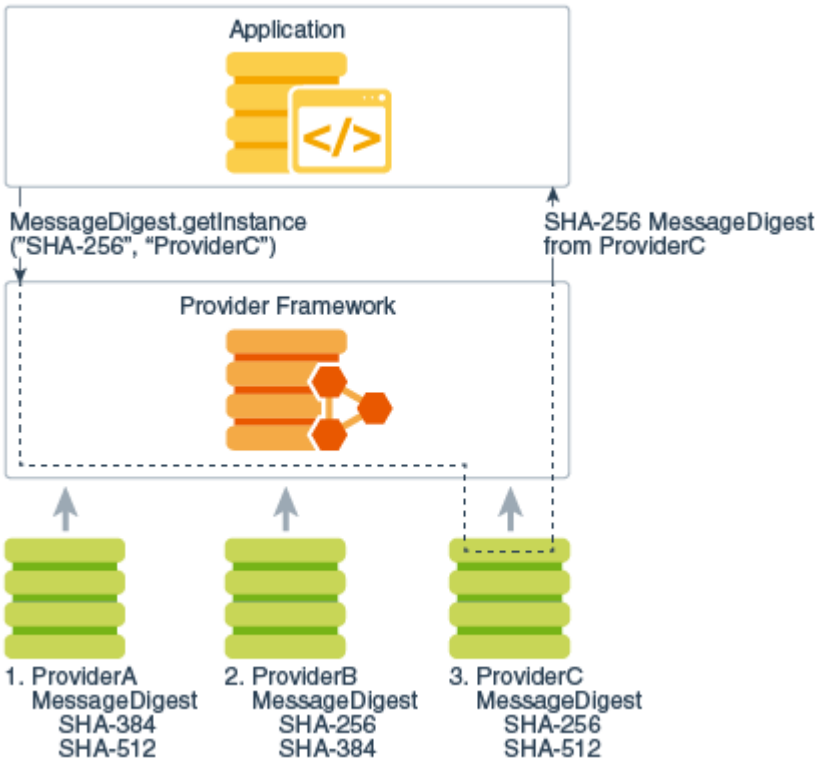
You can optionally request an implementation from a specific provider by specifying the provider's name. The following statement requests a SHA-256 message digest implementation from a specific provider, `ProviderC`:

```
MessageDigest md = MessageDigest.getInstance("SHA-256", "ProviderC");
```

The following figure illustrates how this statement requests a SHA-256 message digest implementation from a specific provider, `ProviderC`. In this case, the implementation from that provider is returned, even though a provider with a higher preference order, `ProviderB`, also supplies a SHA-256 implementation.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 1-2 Request SHA-256 Message Digest Implementation from Specific Provider



For more information about cryptographic services, such as message digest algorithms, see the section Java Cryptography.

Oracle's implementation of the Java platform includes a number of built-in default providers that implement a basic set of security services that can be used by applications. Note that other vendor implementations of the Java platform may include different sets of providers that encapsulate vendor-specific sets of security services. The term built-in default providers refers to the providers available in Oracle's implementation.

Java Cryptography

The Java cryptography architecture is a framework for accessing and developing cryptographic functionality for the Java platform.

It includes APIs for a large variety of cryptographic services, including the following:

- Message digest algorithms
- Digital signature algorithms
- Symmetric bulk and stream encryption
- Asymmetric encryption
- Password-based encryption (PBE)
- Elliptic Curve Cryptography (ECC)
- Key agreement algorithms
- Key Derivation Functions (KDFs)

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Key generators
- Key Encapsulation Mechanisms (KEMs)
- Message Authentication Codes (MACs)
- Secure Random Number Generators

For historical (export control) reasons, the cryptography APIs are organized into two distinct packages:

- The `java.security` and `java.security.*` packages contains classes that are *not* subject to export controls (like `Signature` and `MessageDigest`)
- The `javax.crypto` package contains classes that are subject to export controls (like `Cipher`, `KeyAgreement`, and `KEM`)

The cryptographic interfaces are provider-based, allowing for multiple and interoperable cryptography implementations. Some providers may perform cryptographic operations in software; others may perform the operations on a hardware token (for example, on a smart card device or on a hardware cryptographic accelerator). Providers that implement export-controlled services must be digitally signed by a certificate issued by the Oracle JCE Certificate Authority.

The Java platform includes built-in providers for many of the most commonly used cryptographic algorithms, including the RSA, DSA, and ECDSA signature algorithms, the AES encryption algorithm, the SHA-2 message digest algorithms, and the Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithms. Most of the built-in providers implement cryptographic algorithms in Java code.

The Java platform also includes a built-in provider that acts as a bridge to a native PKCS#11 (v2.x) token. This provider, named `SunPKCS11`, allows Java applications to seamlessly access cryptographic services located on PKCS#11-compliant tokens.

On Windows, the Java platform includes a built-in provider that acts as a bridge to the native Microsoft CryptoAPI. This provider, named `SunMSCAPI`, allows Java applications to seamlessly access cryptographic services on Windows through the CryptoAPI.

Public Key Infrastructure

Public Key Infrastructure (PKI) is a term used for a framework that enables secure exchange of information based on public key cryptography. It allows identities (of people, organizations, etc.) to be bound to digital certificates and provides a means of verifying the authenticity of certificates. PKI encompasses keys, certificates, public key encryption, and trusted Certification Authorities (CAs) who generate and digitally sign certificates.

The Java platform includes APIs and provider support for X.509 digital certificates and Certificate Revocation Lists (CRLs), as well as PKIX-compliant certification path building and validation. The classes related to PKI are located in the `java.security` and `java.security.cert` packages.

Key and Certificate Storage

The Java platform provides for long-term persistent storage of cryptographic keys and certificates via key and certificate stores. Specifically, the `java.security.KeyStore` class represents a *key store*, a secure repository of cryptographic keys and/or trusted certificates (to be used, for example, during certification path validation), and the `java.security.cert.CertStore` class represents a *certificate store*, a public and potentially vast repository of unrelated and typically untrusted certificates. A `CertStore` may also store CRLs.

Copyright © 1993, 2026, Oracle and/or its affiliates.

`KeyStore` and `CertStore` implementations are distinguished by types. The Java platform includes the standard PKCS11 and PKCS12 key store types (whose implementations are compliant with the corresponding PKCS specifications from the Internet Engineering Task Force (IETF)). It also contains a proprietary file-based key store type called JKS (which stands for Java Key Store), and a type called DKS (Domain Key Store) which is a collection of keystores that are presented as a single logical keystore.

> **Note**
>
> JKS and JCEKS keystores will be removed in a future release because they use outdated cryptographic algorithms. It is advised that you migrate to a PKCS12 keystore by using the `keytool -importkeystore` command.

The Java platform includes a special built-in key store, `cacerts`, that contains a number of certificates for well-known, trusted CAs. The keytool utility is able to list the certificates included in `cacerts`. See `keytool` in *Java Development Kit Tool Specifications*.

The SunPKCS11 provider mentioned in the section Java Cryptography includes a PKCS11 `KeyStore` implementation. This means that keys and certificates residing in secure hardware (such as a smart card) can be accessed and used by Java applications via the `KeyStore` API. Note that smart card keys may not be permitted to leave the device. In such cases, the `java.security.Key` object returned by the `KeyStore` API may simply be a reference to the key (that is, it would not contain the actual key material). Such a `Key` object can only be used to perform cryptographic operations on the device where the actual key resides.

The Java platform also includes an LDAP certificate store type (for accessing certificates stored in an LDAP directory), as well as an in-memory Collection certificate store type (for accessing certificates managed in a `java.util.Collection` object).

The Java platform supports native Microsoft Windows keystore types. See the algorithm names for the `KeyStore` engine class in The SunMSCAPI Provider. The Java platform also includes a `KeyStore` implementation that proivdes access to the macOS Keychain. See the algorithm names for the `KeyStore` engine class in The Apple Provider.

Public Key Infrastructure Tools

There are two built-in tools for working with keys, certificates, and key stores:

- `keytool` creates and manages key stores. Use it to perform the following tasks:
  - Create public/private key pairs
  - Display, import, and export X.509 v1, v2, and v3 certificates stored as files
  - Create X.509 certificates
  - Issue certificate (PKCS#10) requests to be sent to CAs
  - Create certificates based on certificate requests
  - Import certificate replies (obtained from the CAs sent certificate requests)
  - Designate public key certificates as trusted
  - Accept a password and store it securely as a secret key
- `jarsigner` signs JAR files and verifies signatures on signed JAR files. The Java ARchive (JAR) file format enables the bundling of multiple files into a single file. Typically, a JAR file contains the class files and auxiliary resources associated with applets and applications.

Copyright © 1993, 2026, Oracle and/or its affiliates.

To digitally sign code, perform the following:

1. Use `keytool` to generate or import appropriate keys and certificates into your key store (if they are not there already).
2. Use the `jar` tool to package the code in a JAR file.
3. Use the `jarsigner` tool (or the `jdk.security.jarsigner` API) to sign the JAR file. The `jarsigner` tool accesses a key store to find any keys and certificates needed to sign a JAR file or to verify the signature of a signed JAR file.

> **Note**
>
> `jarsigner` can optionally generate signatures that include a timestamp. Systems that verify JAR file signatures can check the timestamp and accept a JAR file that was signed while the signing certificate was valid rather than requiring the certificate to be current. (Certificates typically expire annually, and it is not reasonable to expect JAR file creators to re-sign deployed JAR files annually.)

See `keytool` and `jarsigner` in *Java Development Kit Tool Specifications*.

Authentication

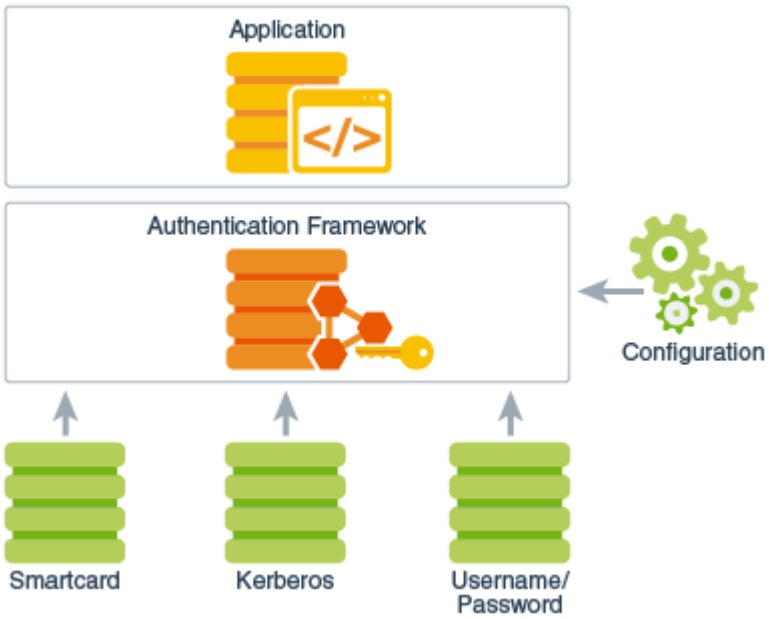
Authentication is the process of determining the identity of a user. In the context of the Java runtime environment, it is the process of identifying the user of an executing Java program. In certain cases, this process may rely on the services described in the section Java Cryptography.

The Java platform provides APIs that enable an application to perform user authentication via pluggable login modules. Applications call into the `LoginContext` class (in the `javax.security.auth.login` package), which in turn references a configuration. The configuration specifies which login module (an implementation of the `javax.security.auth.spi.LoginModule` interface) is to be used to perform the actual authentication.

Since applications solely talk to the standard `LoginContext` API, they can remain independent from the underlying plug-in modules. New or updated modules can be plugged in for an application without having to modify the application itself. The following figure illustrates the independence between applications and underlying login modules:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 1-3 Authentication Login Modules Plugging into the Authentication Framework



It is important to note that although login modules are pluggable components that can be configured into the Java platform, they are not plugged in via security providers. Therefore, they do not follow the provider searching model as described in the section Security Providers. Instead, as is shown in Figure 1-3, login modules are administered by their own unique configuration.

The Java platform provides the following built-in login modules, all in the `com.sun.security.auth.module` package:

- `JndiLoginModule` for username/password authentication using LDAP or NIS databases
- `KeyStoreLoginModule` for logging into any type of key store, including a PKCS#11 token key store
- `Krb5LoginModule` for authentication using Kerberos protocols
- `LdapLoginModule` for LDAP-based authentication
- `NTLoginModule` for authentication using a user's Windows NT security information
- `UnixLoginModule` for authentication using a user's UNIX `Principal` information

Authentication can also be achieved during the process of establishing a secure communication channel between two peers. The Java platform provides implementations of a number of standard communication protocols, which are discussed in the section Secure Communication.

Secure Communication

The data that travels across a network can be accessed by someone who is not the intended recipient. When the data includes private information, such as passwords and credit card numbers, steps must be taken to make the data unintelligible to unauthorized parties. It is also important to ensure that you are sending the data to the appropriate party, and that the data has not been modified, either intentionally or unintentionally, during transport.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Cryptography forms the basis required for secure communication; see the section Java Cryptography. The Java platform also provides API support and provider implementations for a number of standard secure communication protocols.

TLS and DTLS Protocols

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols which provide a secure channel between two communication peers. TLS uses a combination of cryptographic processes by providing authentication, confidentiality and integrity properties for communication over a untrusted or potential hostile network. TLS runs over a reliable, stream-oriented transport channel, typically Transport Control Protocol (TCP). TLS is application protocol independent. Higher-level protocols, for example Hypertext Transfer Protocol (HTTP), can layer on top of TLS transparently.

The Datagram Transport Layer Security (DTLS) protocols are based on the stream-oriented TLS protocols and are intended to provider similar security properties for datagram transport, like User Datagram Protocol (UDP), which does not provide reliable or in-order delivery of data.

The JDK provides APIs and an implementation of the SSL, TLS, and DTLS protocols that includes functionality for data encryption, message integrity, and server and client authentication. Applications can use (D)TLS to provide for the secure passage of data between two peers over any application protocol, such as HTTP on top of TCP/IP.

The `javax.net.ssl.SSLSocket` class represents a network socket that encapsulates TLS support on top of a normal stream socket (`java.net.Socket`). Some applications might want to use alternate data transport abstractions (for example, New-I/O); the `javax.net.ssl.SSLEngine` class is available to produce and consume TLS/DTLS packets.

The JDK also includes APIs that support the notion of pluggable (provider-based) key managers and trust managers. *A key manager* is encapsulated by the `javax.net.ssl.KeyManager` class, and manages the keys used to perform authentication. A *trust manager* is encapsulated by the `TrustManager` class (in the same package), and makes decisions about who to trust based on certificates in the key store it manages.

The JDK includes a built-in provider that implements the SSL/TLS/DTLS protocols:

- SSL 3.0
- TLS 1.0
- TLS 1.1
- TLS 1.2
- TLS 1.3
- DTLS 1.0
- DTLS 1.2

Simple Authentication and Security Layer (SASL)

Simple Authentication and Security Layer (SASL) is an Internet standard that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged, but does not itself specify the contents of that data. It is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit. There are a number of standard SASL mechanisms defined by the Internet community for various security levels and deployment scenarios.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Java SASL API, which is in the `java.security.sasl` module, defines classes and interfaces for applications that use SASL mechanisms. It is defined to be mechanism-neutral; an application that uses the API need not be hardwired into using any particular SASL mechanism. Applications can select the mechanism to use based on desired security features. The API supports both client and server applications. The `javax.security.sasl.Sasl` class is used to create `SaslClient` and `SaslServer` objects.

SASL mechanism implementations are supplied in provider packages. Each provider may support one or more SASL mechanisms and is registered and invoked via the standard provider architecture.

The Java platform includes a built-in provider that implements the following SASL mechanisms:

- CRAM-MD5, DIGEST-MD5, EXTERNAL, GSSAPI, NTLM, and PLAIN client mechanisms
- CRAM-MD5, DIGEST-MD5, GSSAPI, and NTLM server mechanisms

Generic Security Service API and Kerberos

The Java platform contains an API with the Java language bindings for the Generic Security Service Application Programming Interface (GSS-API), which is in the java.security.jgss module. GSS-API offers application programmers uniform access to security services atop a variety of underlying security mechanisms. The Java GSS-API currently requires use of a Kerberos v5 mechanism, and the Java platform includes a built-in implementation of this mechanism. At this time, it is not possible to plug in additional mechanisms.

> **Note**
>
> The `Krb5LoginModule` mentioned in the section Authentication can be used in conjunction with the GSS Kerberos mechanism.

The Java platform also includes a built-in implementation of the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) GSS-API mechanism.

Before two applications can use GSS-API to securely exchange messages between them, they must establish a joint security context. The context encapsulates shared state information that might include, for example, cryptographic keys. Both applications create and use an `org.ietf.jgss.GSSContext` object to establish and maintain the shared information that makes up the security context. Once a security context has been established, it can be used to prepare secure messages for exchange.

The Java GSS APIs are in the `org.ietf.jgss` package. The Java platform also defines basic Kerberos classes, like `KerberosPrincipal`, `KerberosTicket`, `KerberosKey`, and `KeyTab`, which are located in the `javax.security.auth.kerberos` package.

XML Signature

The Java XML Digital Signature API is a standard Java API for generating and validating XML Signatures.

XML Signatures can be applied to data of any type, XML or binary (see XML Signature Syntax and Processing). The resulting signature is represented in XML. An XML Signature can be used to secure your data and provide data integrity, message authentication, and signer authentication.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The API is designed to support all of the required or recommended features of the W3C Recommendation for XML-Signature Syntax and Processing. The API is extensible and pluggable and is based on the Java Cryptography Service Provider Architecture.

The Java XML Digital Signature API, which is in the `java.xml.crypto` module, consists of six packages:

- `javax.xml.crypto`
- `javax.xml.crypto.dsig`
- `javax.xml.crypto.dsig.keyinfo`
- `javax.xml.crypto.dsig.spec`
- `javax.xml.crypto.dom`
- `javax.xml.crypto.dsig.dom`

Java API for XML Processing (JAXP)

Java API for XML Processing (JAXP) is for processing XML data using Java applications. It includes support for Simple API for XML (SAX), Document Object Models (DOM) and Streaming API for XML (StAX) parsers, XML Schema Validation, and Extensible Stylesheet Language Transformations (XSLT). In addition, JAXP provides secure processing features that can help safeguard your applications and system from XML-related attacks. See the Java API for XML Processing (JAXP) Security Guide.

> **Note**
>
> Secure Coding Guidelines for Java SE contains additional recommendations that can help defend against XML-related attacks.

Security Tools Summary

The following tables describe Java security and Kerberos-related tools.

Table 1-2 Java Security Tools

Tool	Usage
`jar`	Creates Java Archive (JAR) files
`jarsigner`	Signs and verifies signatures on JAR files
`keytool`	Creates and manages key stores

There are also three Kerberos-related tools that are shipped with the JDK for Windows. Equivalent functionality is provided in tools of the same name that are automatically part of Linux and macOS.

Table 1-3 Kerberos-related Tools

Tool	Usage
`kinit`	Obtains and caches Kerberos ticket-granting tickets

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 1-3 (Cont.) Kerberos-related Tools

Tool	Usage
`klist`	Lists entries in the local Kerberos credentials cache and key table
`ktab`	Manages the names and service keys stored in the local Kerberos key table

The JAR Signing and Verification Tool

The `jarsigner` tool can be used to digitally sign Java archives (JAR files), and to verify such signatures. This tool depends on the keystore that is managed by `keytool`.

> **Note**
>
> You can also use the `jdk.security.jarsigner` API to sign JAR files.

The Key and Certificate Management Tool

`keytool` is a key and certificate management utility. It enables users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user authenticates himself/herself to other users/services) or data integrity and authentication services, using digital signatures. The authentication information includes both a sequence (chain) of X.509 certificates, and an associated private key, which can be referenced by a so-called "alias". This tool also manages certificates (that are "trusted" by the user), which are stored in the same database as the authentication information, and can be referenced by an "alias".

`keytool` stores the keys and certificates in a so-called *keystore*. The default keystore implementation implements the keystore as a file. It protects private keys with a password.

The chains of X.509 certificates are provided by organizations called Certification Authorities, or CAs. Identities (including CAs) use their private keys to authenticate their association with objects (such as with channels which are secured using SSL), with archives of code they signed, or (for CAs) with X.509 certificates they have issued. As a bootstrapping tool, certificates generated using the `-gencert` option may be used until a Certification Authority returns a certificate chain.

The private keys in this database are always stored in encrypted form, to make it difficult to disclose these private keys inappropriately. A password is required to access or modify the database. These private keys are encrypted using the "password", which should be several words long. If the password is lost, those authentication keys cannot be recovered.

In fact, each private key in the keystore can be protected using its own individual password, which may or may not be the same as the password that protects the keystore's overall integrity.

This tool is (currently) intended to be used from the command line, where one simply types `keytool` as a shell prompt. `keytool` is a script that executes the appropriate Java classes and is built together with the SDK.

The command line options for each command may be provided in any order. Typing an incorrect option or typing `keytool -help` will cause the tool's usage to be summarized on the output device (such as a shell window).

Copyright © 1993, 2026, Oracle and/or its affiliates.

Restricting the Java Platform from Reading Passwords from the Standard Input Stream

Tools such as `keytool` and `jarsigner`, as well as `PasswordCallback` objects processed by `TextCallbackHandler`, can prompt for passwords from the standard input stream if a console is not available, typically when standard input is redirected from a redirected file or connected to an interprocess pipe.

If you set the security property `jdk.security.password.allowSystemIn` to `false`, then the Java platform will throw an exception if it attempts to read a password from the standard input stream without a console. The default value for this security property is `true`.

Built-In Providers

The Java SE implementation from Oracle includes a number of built-in provider packages. See JDK Providers Documentation.

The Security Properties File

A security properties file is a text file that contains names of security properties and their values. You can customize certain aspects of Java security by setting these properties.

When you launch a Java application from a JDK located in `$JAVA_HOME`, by default, the JVM will set the set the security properties to the values specified in `$JAVA_HOME/conf/security/java.security`, which is known as the *master security properties file*. It's possible to specify another security properties file; see Specifying an Alternative Security Properties File.

A security property set in a security properties file is *statically* set. You can *dynamically* set security properties by setting their values in your application's code. See Statically Setting a Security Property in a Security Properties File and Dynamically Setting a Security Property in Application Code.

See Troubleshooting Security Properties for information about enabling logging for security properties and viewing them.

You can include text files containing security property settings in the master security properties file or any other security properties file. When you include a file, all of its security property settings are added as if defined at that point. See Including a Security Properties File.

By default, the master security properties file sets security properties that customize certain aspects of Java, which include the following:

- Registering a security provider: A security provider is a package or set of packages that supply a concrete implementation of a subset of the cryptography aspects of the Java Security API. The master security properties file sets several security properties in the form `security.provider.`*n*, where *n* is the provider's preference order. The preference order is the order in which providers are searched for requested algorithms (when no specific provider is requested).

  See Step 8.1: Configure the Provider for more information.

- Algorithm restrictions: This covers restricted and legacy algorithms for certificate path validation, TLS, signed JAR files, and XML signature validations.

  For example, the `jdk.certpath.disabledAlgorithms` and `jdk.tls.disabledAlgorithm` security properties list which algorithms to disable during certification path validation and TLS/DTLS protocol negotiation. In addition, the `jdk.crypto.disabledAlgorithms` security

Copyright © 1993, 2026, Oracle and/or its affiliates.

property disables algorithms associated with certain Java cryptographic services. See Disabled and Restricted Cryptographic Algorithms for more information about these properties.

- Java Secure Socket Extension (JSSE): JSSE enables secure Internet communications. It provides a framework and an implementation for a Java version of the TLS and DTLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Related security properties include:
  - `jdk.tls.keyLimits`, which limits the amount of data algorithms may encrypt with a set of keys
  - `ssl.KeyManagerFactory` and `ssl.TrustManagerFactory`, which specify the default key and trust manager factory algorithms for the `javax.net.ssl` package

  See Customizing JSSE for more information.
- Other aspects of Java security: This includes default keystore type, configuration of `SecureRandom` implementations, and Kerberos.

Specifying an Alternative Security Properties File

You can specify an alternate `java.security` properties file from the command line with the system property `java.security.properties=<URL>`. This properties file is appended to the master security properties file. If you specify a properties file with `java.security.properties==<URL>` (using two equals signs), then that properties file will completely override the master security properties file.

Statically Setting a Security Property in a Security Properties File

To statically set a security property value in a security properties file, add or modify an existing line in the following form:

```
propertyName=propertyValue
```

For example, suppose that you want to specify a different key manager factory algorithm name than the default SunX509. You do this by specifying the algorithm name as the value of a security property named `ssl.KeyManagerFactory.algorithm`. For example, to set the value to MyX509, add the following line:

```
ssl.KeyManagerFactory.algorithm=MyX509
```

To comment out a line in a security properties file, which means the JVM ignores it when it sets security properties from a security properties file, insert the number sign (#) at the beginning of the line.

By default, the master security properties file contains many comments that describe in detail the security properties specified in it. Sometimes, these security properties themselves are commented out. These security properties that are commented out might have a value specified or no value at all.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> A security property that has been set to no value is set to the empty string. A security property that has been commented out is set to a null value. In this case, the security property might be assigned a default value. The comments in the master security properties file should specify whether a security property has a default value.

Dynamically Setting a Security Property in Application Code

To set a security property dynamically in application code, call the `java.security.Security.setProperty` method:

```
Security.setProperty("propertyName," "propertyValue");
```

For example, a call to the `setProperty()` method corresponding to the previous example for specifying the key manager factory algorithm name would be:

```
Security.setProperty("ssl.KeyManagerFactory.algorithm", "MyX509");
```

> **Note**
>
> Some security properties cannot be set dynamically if they have already been read from a security properties file and cached, which happens when the `java.security.Security` class is initialized. No exception will be thrown if your code attempts to do this.

Troubleshooting Security Properties

Enable logging for security properties by specifying the command-line option `-Djava.security.debug=properties`. Messages prefixed by `properties` contain the final values for all security properties and information on how `include` directives have been processed. See The java.security.debug System Property.

The command-line option `-XshowSettings:security` prints an overview of the security settings that are effective in the JDK. See The java -XshowSettings:security Option.

You can use the Java Flight Recorder (JFR) event `jdk.InitialSecurityProperty` to obtain the initial values for security properties on a running JDK.

Including a Security Properties File

To include a security properties file in another security properties file, use an `include` directive. This feature enables security properties to be defined in multiple files and can facilitate the centralized management of security profiles across multiple JDKs.

Copyright © 1993, 2026, Oracle and/or its affiliates.

For example, on Linux and macOS, to include the security properties file `/usr/lib/jvm/jdk-27/conf/security/extra.security` in another security properties file, add the following `include` directive:

```
include /usr/lib/jvm/jdk-27/conf/security/extra.security
```

On Windows, to include the security properties file `C:\Java\jdk-27\conf\security\extra.security` in another security properties file, add the following `include` directive:

```
include C:\\Java\\jdk-27\\conf\\security\\extra.security
```

A file system path in an `include` directive can be absolute, like the previous examples, or relative. When it's a relative path, the location of the file that contains the `include` directive is used as the base. You can't use a relative path if the `include` directive is in a URL stream. For example, if you specify the following option on the command line when launching a Java application, then the security properties file `extra.java.security` can't contain any `include` directives that specify a relative path:

```
-Djava.security.properties=https://example.com/extra.java.security
```

On Linux and macOS, forward slashes (`/`) are supported.

On Windows, double backward slashes (`\\`), single forward slashes (`/`) and UNC paths are supported. For example:

```
include C:\\Program Files\\Java\\jdk-27\\java.config
include C:/Program Files/Java/jdk-27/additional.java.config
include \\\\WindowsHost\\Share\\unc.path.java.config
```

URL file paths (for example, those that start with `file://`) are not supported.

`include` directives support property expansion. This is similar to expanding variables in a shell. When a string like `${some.property}` appears in an `include` directive, it will be expanded to the value of the system property. For example:

```
include ${java.home}/extra.security
```

This will expand `${java.home}` to use the value of the `java.home` system property. If that property's value is `/usr/lib/jvm/jdk-27`, then the previous example is equivalent to:

```
include /usr/lib/jvm/jdk-27/conf/security/extra.security
```

To assist in platform-independent policy files, you can also use the special notation of `${/}`, which is a shortcut for `${file.separator}`. This allows directives like the following:

```
include ${java.home}${/}extra.security
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> If a system property in an `include` directive is undefined, it's replaced with an empty string instead of the JVM generating an error.

When the JVM encounters an `include` directive, it processes it immediately irrespective of preceding or subsequent occurrences. Its position in the file may affect security properties values as described in Order of include Directives in Security Properties Files. However, `include` directives don't interfere with each other; the JVM always processes them. By contrast, security properties defined later in a security properties or brought in by a following inclusion would override a previous definition if their names are the same.

The JVM generates an error if a file can't be included. This may happen if the file can't be resolved, doesn't exist, is a directory, has insufficient permissions for the JVM to read it, is recursively included more than once, or for any other reason.

Included security properties files can include others recursively as long as it doesn't lead to a cycle. For example, suppose that `java.security` includes the security properties file `A.security`, and `A.security` includes the security properties file `B.security`. The security properties file `B.security` can't include `A.security` as this would create a cycle.

Order of include Directives in Security Properties Files

When you include a security properties file with an `include` directive, properties defined in the included file may either be added to the map of security properties or replace existing ones if their names are the same. Thus, the effect of an `include` directive may depend on its position in the properties file. Consequently, you can specify overridable security properties by defining them before an `include` directive, and specify non-overridable security properties by defining them after an `include` directive.

Security Considerations and include Directives in Security Properties Files

It's important that the layout of your security properties files enable system administrators to clearly and easily understand the reasoning why security properties have been defined and overridden. It's a good practice to check the final map of security properties in the JDK with the methods described in Troubleshooting Security Properties.

> **Note**
>
> When including a security properties file, it's strongly recommended to verify that it comes from a trusted source and to check write permissions not only for the included file but also for the entire chain of parent directories starting at the file system's root. Any user who has capabilities to modify an included file may downgrade the security of the JDK by either replacing a security properties file or adding new ones that cause security property values to be overridden. Any user that can rename a directory in the chain of parent directories may be able to impersonate the included file with a rogue version, leading to a downgrade in the JDK security. As a rule of thumb, an included file and its chain of parent directories should have equal or more restrictive permissions than the file containing its `include` directive.

When placeholders are used in file system paths and system properties are passed as described in Security Profiles, it's worth noting the impact of a typo: If the typo is in the property

Copyright © 1993, 2026, Oracle and/or its affiliates.

name, then the placeholder will be ignored. If the typo is in the property value, an error will be thrown because included files must exist. Always check system properties in placeholders and arguments for typos.

Examples of Using include Directives

Security Profiles

System administrators frequently need to enforce security profiles across Java applications in multiple JDK deployments for security hardening, regulatory compliance, or preserving backward compatibility. The security properties specified in these security profiles should be in addition to those specified by individual JDK deployments. Often, system administrators need to enforce different security profiles depending on the environment, such as development or production. They also should be able to manage security profiles in a centralized location.

To implement this scenario, define multiple *global security profiles* through separate security property files. A global security profile specifies security properties that are required in each JDK deployment. Define a *bridge file*, which contains an `include` directive that specifies the global security profile enforced by default. Include this bridge file in the master security properties file of each JDK deployment. To change the default profile, modify the `include` directive in the bridge file.

To illustrate this concept of managing multiple global security profiles through a bridge file, consider the following layout of security property files and their `include` directives:

- Directory `/global/profiles`
  - File `profile-.security` (bridge file):

    ```
    include profile-prod.security
    ```

  - File `profile-custom.security` (global security profile)
  - File `profile-dev.security` (global security profile)
  - File `profile-prod.security` (global security profile, which is enforced by default)
- JDK deployment A: directory `$JAVA_HOME_DEPLOYMENT_A/conf/security`
  - File `java.security`:

    ```
    include /global/profiles/profile-${securityProfile}.security
    ```

- JDK deployment B: directory `$JAVA_HOME_DEPLOYMENT_B/conf/security`
  - File `java.security`:

    ```
    include /global/profiles/profile-${securityProfile}.security
    ```

In this example layout, `profile-.security` is the bridge file that contains an `include` directive that specifies the global security profile enforced by default, `profile-prod.security` . A system administrator can change the default global security profile by modifying the `include` directive in the bridge file and referring to any of the policies in the directory `/global/profiles` (`profile-custom.security`, `profile-dev.security`, or `profile-prod.security`).

On some occasions, applications require a security profile different from the one enforced by default. In this example, `include` directives in the JDK deployments A and B use the

Copyright © 1993, 2026, Oracle and/or its affiliates.

placeholder suffix `securityProfile`. When the system property `securityProfile` is undefined, this placeholder is replaced with an empty string, which results in the JDK deployments A and B using the bridge file `profile-.security`. When the system property `securityProfile` is set to one of the available values (`custom`, `dev`, or `prod`), the respective global security profile is selected. For example, if you launch a Java application with the `-DsecurityProfile=dev` command-line option, the `profile-dev.security` global security profile would be applied in both JDK deployments A and B, irrespective of the profile enforced by default.

A layout in which the `profile-.security` file doesn't exist prevents a default choice and forces `securityProfile` to be defined.

> **Note**
>
> Security properties are processed when the `java.security.Security` class is initialized. If a placeholder's system property is programmatically set after this point (by calling the `java.lang.System.setProperty(String key, String value)` method) to specify a global security profile, it will have no effect.

You can extend this example to include security profiles that are specific to a JDK release by creating security properties files named `profile-.27.security` and `profile-prod.27.security`. In this case, the `java.security` security properties file for JDK 27 must include both the global and the release-specific bridge files in addition to the current global profile. Having a release-specific profile may be useful when the map of security properties is not the same across all JDK releases.

The JDK neither provides curated security profiles nor favors any particular structure for their organization. It's important to keep profiles updated and opt for secure defaults whenever possible. It's also recommended to analyze how profiles apply to each JDK release in terms of the availability of security properties.

Standard Algorithm Names

See Java Security Standard Algorithm Names Specification for information about the set of standard names for algorithms, certificate and keystore types that Java SE requires and uses.

The Security Manager Is Permanently Disabled

As of JDK 24, the Security Manager is permanently disabled.

The following is a summary of the changes made to the JDK as a result of the Security Manager being permanently disabled:

- You can't enable the Security Manager at startup, nor can you install a custom Security Manager during run time. See Enabling the Security Manager Is an Error.
- Methods that are part of the Security Manager API have been degraded to either return `null` or `false`, pass through the caller's request, or unconditionally throw a `SecurityException` or `UnsupportedOperationException`. See Changes to the Security Manager API.
- Many other APIs are changed, in particular, those that throw a `SecurityException`. See Other Changes to APIs.
- System properties and security properties specific to the Security Manager are ignored. See Affected System and Security Properties.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- The system policy file, `$JAVA_HOME/conf/security/java.policy`, is removed.
- Changes not directly related to API behavior, system or security properties, or changes to the JDK files are listed in Other Changes.

The Security Manager API will be removed in a future JDK release.

See Migrating Away From the Security Manager for tips on verifying if your application requires a Security Manager and migrating away from it.

See JEP 486: Permanently Disable the Security Manager for more information, including an explanation of why the Security Manager is permanently disabled and will be removed in a future release.

Enabling the Security Manager Is an Error

You can't enable the Security Manager nor install a custom Security Manager at startup, for example:

- `java -Djava.security.manager -jar app.jar`
- `java -Djava.security.manager="" -jar app.jar`
- `java -Djava.security.manager=allow -jar app.jar`
- `java -Djava.security.manager=default -jar app.jar`
- `java -Djava.security.manager=com.example.CustomSecurityManager -jar app.jar`

Attempting to do so causes the JVM to report the error and exit, for example:

```
Error occurred during initialization of VM
java.lang.Error: A command line option has attempted to allow or enable the
        Security Manager. Enabling a Security Manager is not supported.
        at java.lang.System.initPhase3(java.base@24/System.java:2067)
```

You can't suppress this error message nor change it to a warning.

You also can't install a Security Manager at run time by calling `System.setSecurityManager(SecurityManager)`. Attempting to do so causes the JVM to throw an `UnsupportedOperationException` with the following message:

```
Setting a Security Manager is not supported
```

Changes to the Security Manager API

The Security Manager API consists of the following:

- Methods in the `java.lang.SecurityManager` class
- Methods in the `AccessController`, `AccessControlContext`, `Policy`, and `ProtectionDomain` classes of the `java.security` package
- The `getSecurityManager` and `setSecurityManager` methods in the `java.lang.System` class

As a result of the Security Manager being permanently disabled, these methods, as appropriate, return `null` or `false`, pass through the caller's request, or unconditionally throw a `SecurityException` or `UnsupportedOperationException`.

The removal of the Security Manager API is planned for a future release.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The following table lists the changes to the Security Manager API in detail:

Table 1-4 APIs Affected by the Security Manager Being Permanently Disabled

Class	Affected APIs
`java.lang.SecurityManager`	The `check*` methods always throw a `SecurityException`. The `getSecurityContext` method returns an `AccessControlContext` that grants no permissions. In particular, the `checkPermission` method throws an `AccessControlException` and the `getDomainCombiner` method returns `null`.
`java.lang.System`	The `setSecurityManager` method always throws an `UnsupportedOperationException`. The `getSecurityManager` method always returns `null`.
`java.lang.ClassLoader`	The default domain assigned to classes by the `defineClass(String, byte[], int, int)` method is the same as before but it's not granted any permissions. In particular, the `ProtectionDomain.getPermissions()` method always returns `null`.
`java.security.AccessController`	The `doPrivileged` methods (6 variants) and the `doPrivilegedWithCombiner` methods (4 variants) execute the action immediately and behave as if a Security Manager has not been enabled. The `checkPermission` method always throws an `AccessControlException`. The `getContext` method returns an `AccessControlContext` that grants no permissions. In particular, the `checkPermission` method throws an `AccessControlException`, and the `getDomainCombiner` method returns null.
`java.security.AccessControlContext`	The `checkPermission` method always throws a `SecurityException`.
`java.security.Permission`	The `checkGuard` method always throws a `SecurityException`.
`java.security.ProtectionDomain`	A `ProtectionDomain` won't be granted any permissions from the current policy because the current policy is always a `Policy` object that grants no permissions.
`java.security.Policy`	The `setPolicy` method always throws an `UnsupportedOperationException`. The `getPolicy` method returns a `Policy` object that grants no permissions. In particular, the `getParameters` method returns `null`, the `getPermissions` method returns a read-only empty `PermissionCollection`, and the `implies` method returns `false`.
`javax.security.auth.Subject`	The `getSubject` method always throws an `UnsupportedOperationException`. The `doAs` methods (2 variants) and the `doAsPrivileged` methods (2 variants) launch the action and bind the subject to the period of execution. See Migrating from the Deprecated for Removal Subject.getSubject and Subject.doAs Methods to the Subject.current and Subject.callAs Methods.
`java.rmi.RMISecurityManager`	All of the changes to the `java.lang.SecurityManager` class also apply to this subclass.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 1-4 (Cont.) APIs Affected by the Security Manager Being Permanently Disabled

Class	Affected APIs
`java.rmi.server.RMIClassLoader`	The `getSecurityContext` method always returns null.

Other Changes to APIs

- Many methods and constructors were specified to throw a `SecurityException` if a Security Manager was enabled and appropriate permissions had not been granted. These methods no longer throw a `SecurityException`. They operate as if a Security Manager has not been enabled.
- The following methods do nothing: `java.lang.Thread.checkAccess()`, `java.lang.ThreadGroup.checkAccess()`, and `java.util.logging.LogManager.checkAccess()`.

Affected System and Security Properties

- The following system properties are not supported: `java.security.policy`, `jdk.security.filePermCompat`, `sun.security.policy.utf8`, `sun.security.policy.numcaches`, and `sun.net.maxDatagramSockets`.
- The `access` and `policy` options of the `java.security.debug` system property are removed.
- The following security properties are not supported: `policy.provider`, `policy.url.`*`n`*, `policy.ignoreIdentityScope`, `package.access`, and `package.definition`.

Other Changes

The RMI Remote Code Downloading mechanism is removed. The `java.rmi.server.RMIClassLoader` class has a service provider interface (SPI) that can be configured through a system property. See `RMIClassLoader.getDefaultProviderInstance()` for information about the default SPI implementation. The default provider previously supported a feature called Remote Code Downloading which allowed an RMI client to load classes from a codebase specified by an RMI server, and the other way around. Remote Code Downloading was enabled only when a Security Manager was enabled. With the Security Manager being permanently disabled, the Remote Code Downloading mechanism is removed. As a workaround, you can perform class loading from the codebase in your own SPI implementation.

Migrating Away From the Security Manager

See How to Determine if an Application Enables the Security Manager to verify whether your application is using a Security Manager.

If you determine that your application enables the Security Manager, consider replacing it with technologies outside the JDK such as the following:

- Containers, which are a lightweight and portable form of virtualization technology, allow applications and their dependencies to be packaged and run consistently across different environments.
- Hypervisors, which are also called Virtual Machine Monitors (VMMs), enable multiple operating systems to share the hardware resources offered by a single host computer.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Operating system mechanisms such as the macOS App Sandbox or the Linux secure computing (seccomp) feature.

Like the Security Manager, these technologies can restrict how applications use local and remote resources. For example, they can prevent code from accessing the network to exfiltrate data.

A small number of libraries were designed to use the Security Manager if it is enabled. Some of these may also use advanced parts of the Security Manager API to implement a custom execution environment. See Advice to Maintainers of Libraries That Support the Security Manager.

A small number of applications use the Security Manager not to enforce a security policy but, rather, as a means to intercept calls to the Java Platform API. If your application requires interception, you can use alternative techniques such as source code modification, static code analysis and rewriting, and agent-based dynamic code rewriting at class load time. See An Agent that Blocks Code From Calling System.exit(int) for an example of an agent that uses dynamic code rewriting.

How to Determine if an Application Enables the Security Manager

Try the following techniques to determine if an application enables the Security Manager:

- Check scripts or documentation to see if the application is launched with the Security Manager allowed or enabled through command-line options, or requires policy files to be installed and configured.
- Run the application on a JDK release from 17 to 23 and look for messages on the console warning that the Security Manager will be deprecated and will be removed in a future release. See the Issue warnings section in *JEP 411: Deprecate the Security Manager for Removal*.
- Run the application on a JDK release from 17 to 23 with the command-line option `-Djava.security.manager=disallow`. If the application installs a custom Security Manager through the `System.setSecurityManager(SecurityManager)` method, the JVM will throw an `UnsupportedOperationException`.
- Use the `jdeprscan` tool from a JDK release from 17 to 23 to scan for uses of deprecated Security Manager APIs such as `System.setSecurityManager(SecurityManager)` or `java.security.Policy.setPolicy(Policy)`.

Advice to Maintainers of Libraries That Support the Security Manager

A small number of libraries were designed to use the Security Manager if it is enabled. These libraries typically employ two idioms:

- Calling `System.getSecurityManager()` to check if a Security Manager is enabled and, if so, calling one of the `SecurityManager.checkPermission` methods to check if an operation should be granted or denied:

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    sm.checkPermission(...);
}
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

- Calling one of the `AccessController.doPrivileged` methods to run code with different permissions from the calling code:

```
SomeReturnValue v = AccessController.doPrivileged(() -> {
    // ...
    return theResult;
});
```

In JDK 24 and later, where a Security Manager is never enabled, the `System.getSecurityManager()`method and the `AccessController.doPrivileged` methods behave as they did in prior JDK releases when a Security Manager was not enabled:

- `System.getSecurityManager()` returns `null`
- The six `AccessController.doPrivileged` methods run the given action immediately

Accordingly, the small number of libraries that call these methods will run on JDK 24 and later without change. However, we strongly recommend that new releases of these libraries do not call these methods, which will be removed in a future release.

A very small number of libraries use advanced parts of the Security Manager API to implement a custom execution environment. For example, a library might call `AccessController.checkPermission(Permission)` to enforce its own permission model or call `Policy.setPolicy(Policy)` to make custom Security Managers treat certain resources as off-limits.

In JDK 24 and later, these methods always implement an execution environment that disallows access to all resources. As a result, the methods behave differently than they did in prior JDK releases:

- The `AccessController.checkPermission(Permission)` method always throws `AccessControlException`
- The `Policy.setPolicy(Policy)` method always throws `UnsupportedOperationException`
- The `SecurityManager.check*` methods always throw `SecurityException`

An Agent that Blocks Code From Calling System.exit(int)

An *agent* is a Java program that can alter the code of an application while the application is running. Agents achieve this by transforming the bytecodes of methods when classes are loaded, or by redefining classes after they have been loaded. See the `java.lang.instrument` package for more information about agents.

The example Example 1-1 is an agent that blocks code from calling `System.exit(int)`. The agent declares a `premain` method that is run by the JVM before the `main` method of the application. This method registers a *transformer* that transforms class files as they are loaded from the class path or module path. The transformer rewrites every call to `System.exit(int)` into `throw new RuntimeException("System.exit not allowed")`.

The transformer reads and writes bytecodes in class files using the Class-File API.

Example 1-1 BlockSystemExitAgent.java

```
import java.lang.classfile.*;
import java.lang.classfile.instruction.InvokeInstruction;
import java.lang.constant.ClassDesc;
import java.lang.constant.MethodTypeDesc;
import java.lang.instrument.ClassFileTransformer;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
import java.lang.instrument.Instrumentation;
import java.security.ProtectionDomain;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.function.Predicate;

public class BlockSystemExitAgent {
    /*
     * Before the application starts, register a transformer of class files.
     */
    public static void premain(String agentArgs, Instrumentation inst) {
        var transformer = new ClassFileTransformer() {
            @Override
            public byte[] transform(ClassLoader      loader,
                                    String           className,
                                    Class<?>         classBeingRedefined,
                                    ProtectionDomain protectionDomain,
                                    byte[]           classBytes) {
                if (loader != null && loader !=
ClassLoader.getPlatformClassLoader()) {
                    return blockSystemExit(classBytes);
                } else {
                    return null;
                }
            }
        };
        inst.addTransformer(transformer, true);
    }

    /*
     * Rewrite every invokestatic of System.exit(int) to an athrow of
RuntimeException.
     */
    private static byte[] blockSystemExit(byte[] classBytes) {
        var modified = new AtomicBoolean();
        ClassFile cf = ClassFile.of(ClassFile.DebugElementsOption.DROP_DEBUG);
        ClassModel classModel = cf.parse(classBytes);

        Predicate<MethodModel> invokesSystemExit =
            methodModel -> methodModel.code()
                                      .map(codeModel ->
                                             codeModel.elementStream()
                                                      .anyMatch(BlockSystemExi
tAgent::isInvocationOfSystemExit))
                                      .orElse(false);

        CodeTransform rewriteSystemExit =
            (codeBuilder, codeElement) -> {
                if (isInvocationOfSystemExit(codeElement)) {
                    var runtimeException =
ClassDesc.of("java.lang.RuntimeException");
                    codeBuilder.new_(runtimeException)
                               .dup()
                               .ldc("System.exit not allowed")
                               .invokespecial(runtimeException,
                                   "<init>",
                                   MethodTypeDesc.ofDescriptor("(Ljava/lang/
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
String;)V"),
                                    false)
                                .athrow();
                    modified.set(true);
                } else {
                    codeBuilder.with(codeElement);
                }
            };

        ClassTransform ct =
ClassTransform.transformingMethodBodies(invokesSystemExit, rewriteSystemExit);
        byte[] newClassBytes = cf.transformClass(classModel, ct);
        if (modified.get()) {
            return newClassBytes;
        } else {
            return null;
        }
    }

    private static boolean isInvocationOfSystemExit(CodeElement codeElement) {
        return codeElement instanceof InvokeInstruction i
                && i.opcode() == Opcode.INVOKESTATIC
                && "java/lang/System".equals(i.owner().asInternalName())
                && "exit".equals(i.name().stringValue())
                && "(I)V".equals(i.type().stringValue());
    }
}
```

To use `BlockSystemExitAgent`, you must package it in a JAR file and specifiy it with the `-javaagent` option when starting an application.

Follow these steps to compile, package, and use `BlockSystemExitAgent`:

1. Compile `BlockSystemExitAgent` in the directory `agentclasses`:

   ```
   $ javac -d agentclasses BlockSystemExitAgent.java
   ```

2. Create a JAR file manifest in `agent.mf`:

   ```
   $ cat > agent.mf << EOF
   Premain-Class: BlockSystemExitAgent
   Can-Retransform-Classes: true
   EOF
   ```

3. Create the agent JAR file. Note that there's a period after `-C agentclasses`:

   ```
   $ jar --create --file=BlockSystemExitAgent.jar --manifest=agent.mf -C
   agentclasses .
   ```

4. Run your application (which is packaged in `app.jar` in this example) with the agent JAR file:

   ```
   $ java -javaagent:BlockSystemExitAgent.jar -jar app.jar
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Troubleshooting Security

To monitor security access, you can set the `java.security.debug` system property, which determines what trace messages are printed during execution. To view security properties, security providers, and TLS-related settings, specify the `-XshowSettings:security` option in the `java` command. You can enable debugging in JGSS and other related technologies with various system properties or environment variables.

Topics

- The java.security.debug System Property
- Debug Statement Output Format
- The java -XshowSettings:security Option
- Enabling Debugging in Java Generic Security Services

The java.security.debug System Property

To see a list of all debugging options, use the `help` option as follows. *`MyApp`* is any Java application. The `java` command prints the debugging options and then exits before running *`MyApp`*.

```
java -Djava.security.debug=help MyApp
```

> **Note**
>
> - To use more than one option, separate options with a comma.
> - JSSE also provides dynamic debug tracing support for SSL/TLS/DTLS troubleshooting. See Debugging Utilities.

The following table lists `java.security.debug` options and links to further information about each option. See `java.security.debug` in the JavaDoc API documentation for additional information.

Table 1-5 `java.security.debug` Options

Option	Description	Further Information
`all`	Turn on all the debugging options	None

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 1-5 (Cont.) `java.security.debug` Options

Option	Description	Further Information
`certpath`	Turns on debugging for the PKIX `CertPathValidator` and `CertPathBuilder` implementations. Use the `ocsp` option with the `certpath` option for OCSP protocol tracing. A hexadecimal dump of the OCSP request and response bytes is displayed. You can use the following options with this option: • `ocsp`: Dump OCSP protocol exchanges • `verbose`: A hexadecimal dump of the OCSP request and response bytes is displayed	PKI Programmer's Guide Overview
`configfile`	JAAS (Java Authentication and Authorization Service) configuration file loading	Java Authentication and Authorization Service (JAAS) Reference Guide Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges
`configparser`	JAAS configuration file parsing	Java Authentication and Authorization Service (JAAS) Reference Guide Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges
`gsslogginconfig`	Java GSS (Generic Security Services) login configuration file debugging	Java Generic Security Services: (Java GSS) and Kerberos JAAS and Java GSS-API Tutorial `javax.security.auth.login.Configuration`: A Configuration object is responsible for specifying which `javax.net.ssl.SSLEngine` should be used for a particular application, and in what order the `LoginModules` should be invoked. Appendix B: JAAS Login Configuration File Advanced Security Programming in Java SE Authentication, Secure Communication and Single Sign-On

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 1-5 (Cont.) `java.security.debug` Options

Option	Description	Further Information
`jar`	JAR file verification	Verifying Signed JAR Files from The Java Tutorials **Note** Use the System property `jdk.jar.maxSignatureFileSize` to specify the maximum size, in bytes, of signature files in a signed JAR. Its default value is `16000000` (16 MB).
`jca`	JCA engine class debugging	Engine Classes and Algorithms
`KeyStore`	Keystore debugging	Keystores `KeyStore`
`logincontext`	`LoginContext` results	Java Authentication and Authorization Service (JAAS) Reference Guide Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges
`pcsc`	Java Smart Card I/O and SunPCSC provider debugging	The SunPCSC Provider and the `javax.smartcardio` package
`pkcs11`	PKCS11 session manager debugging	PKCS#11 Reference Guide
`pkcs11keystore`	PKCS11 KeyStore debugging	PKCS#11 Reference Guide
`pkcs12`	PKCS12 KeyStore debugging	None
`properties`	`java.security` configuration file debugging	None

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 1-5 (Cont.) `java.security.debug` Options

Option	Description	Further Information
`provider`	Security provider debugging You can use the `engine=<engines>` option with this option. It displays output only for the JCA engines specified by `<engines>`. This value is a comma-separated list. It can contain one or more of the following values: • `Cipher` • `KDF` • `KeyAgreement` • `KeyGenerator` • `KeyPairGenerator` • `KeyStore` • `Mac` • `MessageDigest` • `SecureRandom` • `Signature`	Java Cryptography Architecture (JCA) Reference Guide
`securerandom`	SecureRandom debugging	The SecureRandom Class
`sunpkcs11`	SunPKCS11 provider debugging	PKCS#11 Reference Guide
`ts`	Timestamping debugging	None
`x509`	X.509 certificate debugging You can use the `ava` option with this option. It embeds non-printable/non-escaped characters in AVA components as hex strings	X.509 Certificates and Certificate Revocation Lists (CRLs)

Debug Statement Output Format

Each debug output statement generated through the `java.security.debug` option is formatted as follows:

`componentValue``[`*`threadId`*`|`*`threadName`*`|`*`sourceCodeLocation`*`|`*`timestamp`*`]: <`*`debug statement`*`>`

- *`componentValue`* is the security component value being logged.
- *`threadId`* is the hexadecimal value of the thread ID.
- *`threadName`* is the name of the thread running the log statement.
- *`sourceCodeLocation`* is the source file and line number making this log call in the format *`filename`*`:`*`lineNumber`*.
- *`timestamp`* is the date and time in the format *`yyyy-MM-dd kk:mm:ss.SSS`*.
- `<`*`debug statement`*`>` corresponds to the debug output from the security component.

The java -XshowSettings:security Option

You can specify the option `-XshowSettings:security` option in the `java` command to view security properties, security providers, and TLS-related settings. The option shows third-party

Copyright © 1993, 2026, Oracle and/or its affiliates.

security provider details if they are included in the application class path or module path and such providers are configured in the `java.security` file.

In addition, you can specify `-XshowSettings:security:<subcategory>` where `<subcategory>` is one of the following:

- `all`: show all security settings
- `properties`: show security properties
- `providers`: show static security provider settings
- `tls`: show TLS-related security settings

Enabling Debugging in Java Generic Security Services

Set the following system properties or environment variables to `true` to enable debugging in the Java Generic Security Services (JGSS) framework, Kerberos, SPNEGO, the native JGSS bridge, and the SSPI bridge on Windows:

> **⚠ Caution**
>
> Debugging information may contain sensitive information.

Table 1-6 JGSS Debugging System Properties

System Property or Environment Variable	JGSS Feature to Debug
`sun.security.jgss.debug` system property	JGSS framework
`sun.security.krb5.debug` system property	Java Kerberos 5 mechanism
`sun.security.spnego.debug` system property	Java SPNEGO mechanism
`sun.security.nativegss.debug` system property	Native JGSS bridge
`SSPI_BRIDGE_TRACE` environment variable	SSPI bridge on Windows

For example, to enable debugging in the JGSS framework, set the `sun.security.jgss.debug` system property to `true` on the command line as follows:

```
java -Dsun.security.jgss.debug=true MyApp
```

In your JAAS login configuration file, you can specify `debug=true` in the Krb5LoginModule to enable debugging in the associated entry. For example, the following enables debugging for `JassSample`:

```
JaasSample {
    com.sun.security.auth.module.Krb5LoginModule required;
    debug=true;
};
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

2

Java Cryptography Architecture (JCA) Reference Guide

The Java Cryptography Architecture (JCA) is a major piece of the platform, and contains a "provider" architecture and a set of APIs for digital signatures, message digests (hashes), certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), key generation and management, and secure random number generation, to name a few.

Introduction to Java Cryptography Architecture

The Java platform strongly emphasizes security, including language safety, cryptography, public key infrastructure, authentication, and secure communication.

The JCA is a major piece of the platform, and contains a "provider" architecture and a set of APIs for digital signatures, message digests (hashes), certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), key generation and management, and secure random number generation, to name a few. These APIs allow developers to easily integrate security into their application code. The architecture was designed around the following principles:

- **Implementation independence**: Applications do not need to implement security algorithms. Rather, they can request security services from the Java platform. Security services are implemented in providers (see Cryptographic Service Providers), which are plugged into the Java platform via a standard interface. An application may rely on multiple independent providers for security functionality.
- **Implementation interoperability**: Providers are interoperable across applications. Specifically, an application is not bound to a specific provider, and a provider is not bound to a specific application.
- **Algorithm extensibility**: The Java platform includes a number of built-in providers that implement a basic set of security services that are widely used today. However, some applications may rely on emerging standards not yet implemented, or on proprietary services. The Java platform supports the installation of custom providers that implement such services.

Other cryptographic communication libraries available in the JDK use the JCA provider architecture, but are described elsewhere. The JSSE components provides access to Secure Socket Layer (SSL), Transport Layer Security (TLS), and Datagram Transport Layer Security (DTLS) implementations; see Java Secure Socket Extension (JSSE) Reference Guide. You can use Java Generic Security Services (JGSS) (via Kerberos) APIs, and Simple Authentication and Security Layer (SASL) to securely exchange messages between communicating applications; see Introduction to JAAS and Java GSS-API Tutorials and Java SASL API Programming and Deployment Guide.

Notes on Terminology

- Prior to JDK 1.4, the JCE was an unbundled product, and as such, the JCA and JCE were regularly referred to as separate, distinct components. As JCE is now bundled in the JDK, the distinction is becoming less apparent. Since the JCE uses the same architecture as the JCA, the JCE should be more properly thought of as a part of the JCA.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- The JCA within the JDK includes two software components:
  - The framework that defines and supports cryptographic services for which providers supply implementations. This framework includes packages such as `java.security`, `javax.crypto`, `javax.crypto.spec`, and `javax.crypto.interfaces`.
  - The actual providers such as `Sun`, `SunRsaSign`, `SunJCE`, which contain the actual cryptographic implementations.

  Whenever a specific JCA provider is mentioned, it will be referred to explicitly by the provider's name.

> **Warning**
>
> The JCA makes it easy to incorporate security features into your application. However, this document does not cover the theory of security/cryptography beyond an elementary introduction to concepts necessary to discuss the APIs. This document also does not cover the strengths/weaknesses of specific algorithms, not does it cover protocol design. Cryptography is an advanced topic and one should consult a solid, preferably recent, reference in order to make best use of these tools.
> You should always understand what you are doing and why: DO NOT simply copy random code and expect it to fully solve your usage scenario. Many applications have been deployed that contain significant security or performance problems because the wrong tool or algorithm was selected.

JCA Design Principles

The JCA was designed around these principles:

- Implementation independence and interoperability
- Algorithm independence and extensibility

Implementation independence and algorithm independence are complementary; you can use cryptographic services, such as digital signatures and message digests, without worrying about the implementation details or even the algorithms that form the basis for these concepts. While complete algorithm-independence is not possible, the JCA provides standardized, algorithm-specific APIs. When implementation-independence is not desirable, the JCA lets developers indicate a specific implementation.

Algorithm independence is achieved by defining types of cryptographic "engines" (services), and defining classes that provide the functionality of these cryptographic engines. These classes are called *engine classes*, and examples are the `MessageDigest`, `Signature`, `KeyFactory`, `KeyPairGenerator`, and `Cipher` classes.

Implementation independence is achieved using a "provider"-based architecture. The term Cryptographic Service Provider (CSP), which is used interchangeably with the term "provider," (see Cryptographic Service Providers) refers to a package or set of packages that implement one or more cryptographic services, such as digital signature algorithms, message digest algorithms, and key conversion services. A program may simply request a particular type of object (such as a `Signature` object) implementing a particular service (such as the DSA signature algorithm) and get an implementation from one of the installed providers. If desired, a program may instead request an implementation from a specific provider. Providers may be updated transparently to the application, for example when faster or more secure versions are available.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Implementation interoperability means that various implementations can work with each other, use each other's keys, or verify each other's signatures. This would mean, for example, that for the same algorithms, a key generated by one provider would be usable by another, and a signature generated by one provider would be verifiable by another.

Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can be added easily.

Provider Architecture

Providers contain a package (or a set of packages) that supply concrete implementations for the advertised cryptographic algorithms.

Cryptographic Service Providers

`java.security.Provider` is the base class for all security providers. Each CSP contains an instance of this class which contains the provider's name and lists all of the security services/ algorithms it implements. When an instance of a particular algorithm is needed, the JCA framework consults the provider's database, and if a suitable match is found, the instance is created.

Providers contain a package (or a set of packages) that supply concrete implementations for the advertised cryptographic algorithms. Each JDK installation has one or more providers installed and configured by default. Additional providers may be added statically or dynamically. Clients may configure their runtime environment to specify the provider *preference order*. The preference order is the order in which providers are searched for requested services when no specific provider is requested.

To use the JCA, an application simply requests a particular type of object (such as a `MessageDigest`) and a particular algorithm or service (such as the "SHA-256" algorithm), and gets an implementation from one of the installed providers. For example, the following statement requests a SHA-256 message digest from an installed provider:

```
md = MessageDigest.getInstance("SHA-256");
```

Alternatively, the program can request the objects from a specific provider. Each provider has a name used to refer to it. For example, the following statement requests a SHA-256 message digest from the provider named ProviderC:

```
md = MessageDigest.getInstance("SHA-256", "ProviderC");
```

The following figures illustrates requesting an SHA-256 message digest implementation. They show three different providers that implement various message digest algorithms (SHA-256, SHA-384, and SHA-512). The providers are ordered by preference from left to right (1-3). In Figure 2-1, an application requests a SHA-256 algorithm implementation **without** specifying a provider name. The providers are searched in preference order and the implementation from the first provider supplying that particular algorithm, ProviderB, is returned. In Figure 2-2, the application requests the SHA-256 algorithm implementation **from a specific provider**, ProviderC. This time, the implementation from ProviderC is returned, even though a provider with a higher preference order, ProviderB, also supplies an SHA-256 implementation.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 2-1 Request SHA-256 Message Digest Implementation Without Specifying Provider

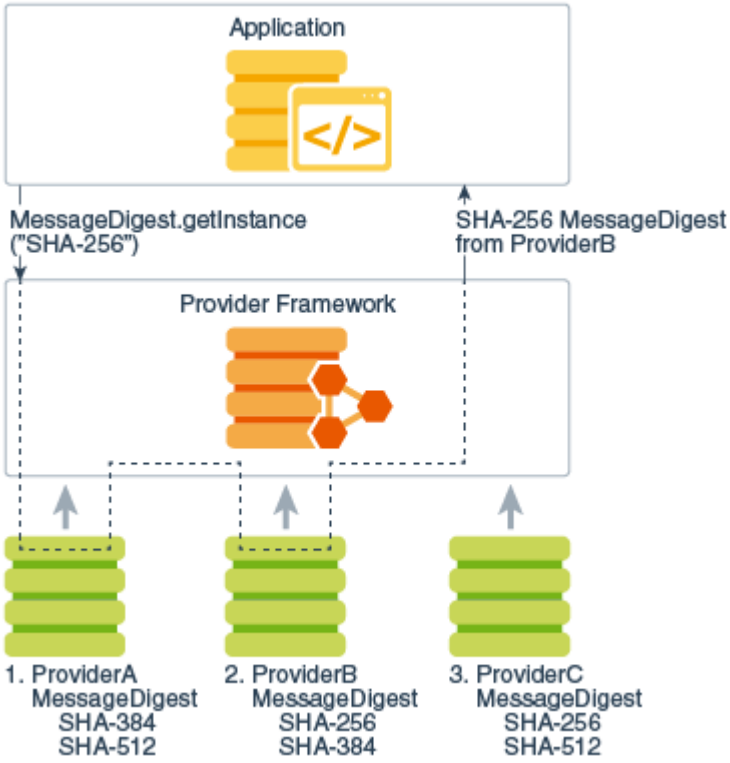
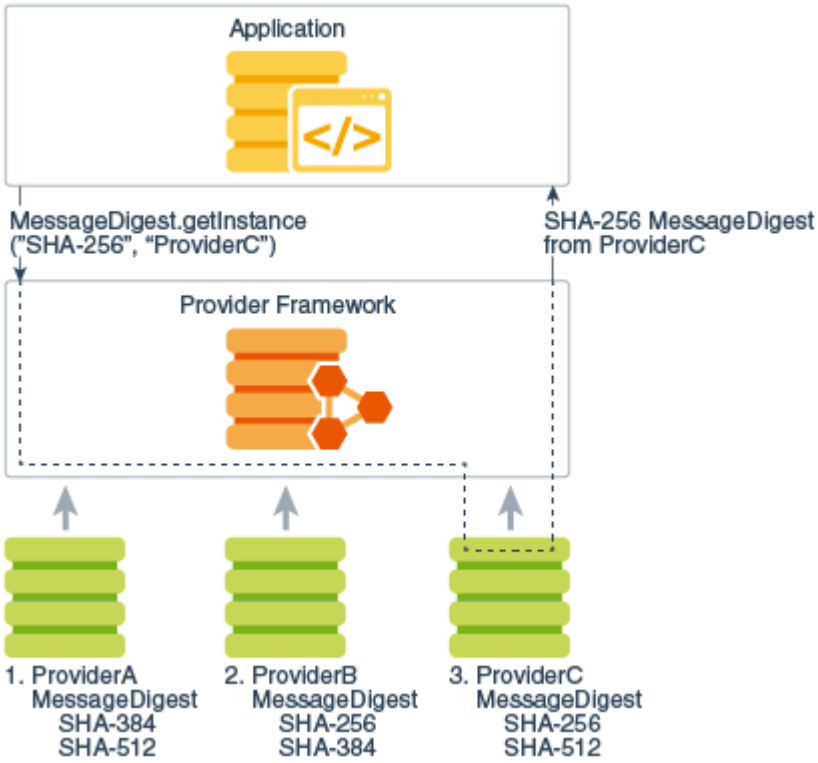


Figure 2-2 Request SHA-256 Message Digest with ProviderC



Copyright © 1993, 2026, Oracle and/or its affiliates.

Cryptographic implementations in the JDK are distributed via several different providers (`Sun`, `SunJSSE`, `SunJCE`, `SunRsaSign`) primarily for historical reasons, but to a lesser extent by the type of functionality and algorithms they provide. Other Java runtime environments may not necessarily contain these providers, so applications should not request a provider-specific implementation unless it is known that a particular provider will be available.

The JCA offers a set of APIs that allow users to query which providers are installed and what services they support.

This architecture also makes it easy for end-users to add additional providers. Many third party provider implementations are already available. See The Provider Class for more information on how providers are written, installed, and registered.

How Providers Are Actually Implemented

Algorithm independence is achieved by defining a generic high-level Application Programming Interface (API) that all applications use to access a service type. *Implementation independence* is achieved by having all provider implementations conform to well-defined interfaces. Instances of engine classes are thus "backed" by implementation classes which have the same method signatures. Application calls are routed through the engine class and are delivered to the underlying backing implementation. The implementation handles the request and returns the proper results.

The application API methods in each engine class are routed to the provider's implementations through classes that implement the corresponding Service Provider Interface (SPI). That is, for each engine class, there is a corresponding abstract SPI class which defines the methods that each cryptographic service provider's algorithm must implement. The name of each SPI class is the same as that of the corresponding engine class, followed by `Spi`. For example, the `Signature` engine class provides access to the functionality of a digital signature algorithm. The actual provider implementation is supplied in a subclass of `SignatureSpi`. Applications call the engine class' API methods, which in turn call the SPI methods in the actual implementation.

Each SPI class is abstract. To supply the implementation of a particular type of service for a specific algorithm, a provider must subclass the corresponding SPI class and provide implementations for all the abstract methods.

For each engine class in the API, implementation instances are requested and instantiated by calling the `getInstance()` factory method in the engine class. A factory method is a static method that returns an instance of a class. The engine classes use the framework provider selection mechanism described previously to obtain the actual backing implementation (SPI), and then creates the actual engine object. Each instance of the engine class encapsulates (as a private field) the instance of the corresponding SPI class, known as the SPI object. All API methods of an API object are declared final and their implementations invoke the corresponding SPI methods of the encapsulated SPI object.

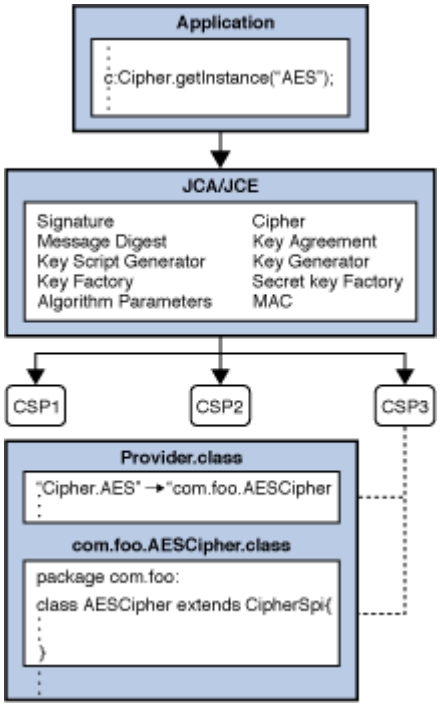
To make this clearer, review Example 2-1 and Figure 2-3:

Example 2-1 Sample Code for Getting an Instance of an Engine Class

```
Cipher c = Cipher.getInstance("AES");
c.init(ENCRYPT_MODE, key);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 2-3 Application Retrieves “AES” Cipher Instance



Here an application wants an "AES" `javax.crypto.Cipher` instance, and doesn't care which provider is used. The application calls the `getInstance()` factory methods of the `Cipher` engine class, which in turn asks the JCA framework to find the first provider instance that supports "AES". The framework consults each installed provider, and obtains the provider's instance of the `Provider` class. (Recall that the `Provider` class is a database of available algorithms.) The framework searches each provider, finally finding a suitable entry in CSP3. This database entry points to the implementation class `com.foo.AESCipher` which extends `CipherSpi`, and is thus suitable for use by the `Cipher` engine class. An instance of `com.foo.AESCipher` is created, and is encapsulated in a newly-created instance of `javax.crypto.Cipher`, which is returned to the application. When the application now does the `init()` operation on the `Cipher` instance, the `Cipher` engine class routes the request into the corresponding `engineInit()` backing method in the `com.foo.AESCipher` class.

Java Security Standard Algorithm Names lists the Standard Names defined for the Java environment. Other third-party providers may define their own implementations of these services, or even additional services.

Keystores

A database called a "keystore" can be used to manage a repository of keys and certificates. Keystores are available to applications that need data for authentication, encryption, or signing purposes.

Applications can access a keystore via an implementation of the `KeyStore` class, which is in the `java.security` package. As of JDK 9, the default and recommended keystore type (format) is "pkcs12", which is based on the RSA PKCS12 Personal Information Exchange Syntax Standard. Previously, the default keystore type was "jks", which is a proprietary format. Other keystore formats are available, such as "jceks", which is an alternate proprietary keystore format, and "pkcs11", which is based on the RSA PKCS11 Standard and supports access to cryptographic tokens such as hardware security modules and smartcards.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> JKS and JCEKS keystores will be removed in a future release because they use outdated cryptographic algorithms. It is advised that you migrate to a PKCS12 keystore by using the `keytool -importkeystore` command.

Applications can choose different keystore implementations from different providers, using the same provider mechanism described previously. See Key Management.

Engine Classes and Algorithms

An engine class provides the interface to a specific type of cryptographic service, independent of a particular cryptographic algorithm or provider.

The engine classes provide one of the following:

- cryptographic operations (for example, encryption, digital signatures, and message digests),
- generators or converters of cryptographic material (keys and algorithm parameters), or
- objects (keystores or certificates) that encapsulate the cryptographic data and can be used at higher layers of abstraction.

The following engine classes are available:

- `SecureRandom`: Used to generate random or pseudo-random numbers.
- `MessageDigest`: Used to calculate the message digest (hash) of specified data.
- `Signature`: Initialized with keys, these are used to sign data and verify digital signatures.
- `Cipher`: Initialized with keys, these used for encrypting/decrypting data. There are various types of algorithms: symmetric bulk encryption (for example, AES), asymmetric encryption (for example, RSA), and password-based encryption (for example, PBE).
- `Mac`: Like `MessageDigest`s, Message Authentication Codes (MACs) also generate hash values, but are first initialized with keys to protect the integrity of messages.
- `KEM`: Used by two parties to derive a shared secret key from a private/public key pair.
- `KeyFactory`: Used to convert existing opaque cryptographic keys of type `Key` into key specifications (transparent representations of the underlying key material) and the other way around.
- `SecretKeyFactory`: Used to convert existing opaque cryptographic keys of type `SecretKey` into key specifications (transparent representations of the underlying key material) and the other way around. `SecretKeyFactory`s are specialized `KeyFactory`s that create secret (symmetric) keys only.
- `KeyPairGenerator`: Used to generate a new pair of public and private keys suitable for use with a specified algorithm.
- `KeyGenerator`: Used to generate new secret keys for use with a specified algorithm.
- `KeyAgreement`: Used by two or more parties to agree upon and establish a specific key to use for a particular cryptographic operation.
- `AlgorithmParameters`: Used to store the parameters for a particular algorithm, including parameter encoding and decoding.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `AlgorithmParameterGenerator`: Used to generate a set of `AlgorithmParameters` suitable for a specified algorithm.
- `KeyStore`: Used to create and manage a *keystore*. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.
- `CertificateFactory`: Used to create public key certificates and Certificate Revocation Lists (CRLs).
- `CertPathBuilder`: Used to build certificate chains (also known as certification paths).
- `CertPathValidator`: Used to validate certificate chains.
- `CertStore`: Used to retrieve `Certificates` and `CRLs` from a repository.

> **Note**
>
> A *generator* creates objects with brand-new contents, whereas a *factory* creates objects from existing material (for example, an encoding).

Core Classes and Interfaces

The following are the core classes and interfaces provided in the JCA.

- `Provider` and `Security`
- `SecureRandom`, `MessageDigest`, `Signature`, `Cipher`, `Mac`, `KDF`, `KEM`, `KeyFactory`, `SecretKeyFactory`, `KeyPairGenerator`, `KeyGenerator`, `KeyAgreement`, `AlgorithmParameter`, `AlgorithmParameterGenerator`, `KeyStore`, and `CertificateFactory` engine classes
- `Key` interfaces and classes, `KeyPair`
- `AlgorithmParameterSpec Interface`, `AlgorithmParameters`, `AlgorithmParameterGenerator`, and algorithm parameter specification interfaces and classes in the `java.security.spec` and `javax.crypto.spec` packages.
- `KeySpec Interface`, `EncodedKeySpec`, `PKCS8EncodedKeySpec`, and `X509EncodedKeySpec`.
- `BinaryEncodable`, `PEMEncoder`, `PEMDecoder`, and `PEM`.

> **Note**
>
> See `CertPathBuilder`, `CertPathValidator`, and `CertStore` engine classes in the Java PKI Programmer's Guide.

The guide will cover the most useful high-level classes first (`Provider`, `Security`, `SecureRandom`, `MessageDigest`, `Signature`, `Cipher`, `Mac`, and `KEM`), then delve into the various support classes. For now, it is sufficient to simply say that Keys (public, private, and secret) are generated and represented by the various JCA classes, and are used by the high-level classes as part of their operation.

This section shows the signatures of the main methods in each class and interface. Examples for some of these classes (`MessageDigest`, `Signature`, `KeyPairGenerator`,

Copyright © 1993, 2026, Oracle and/or its affiliates.

`SecureRandom`, `KeyFactory`, and key specification classes) are supplied in the corresponding Code Examples sections.

The complete reference documentation for the relevant Security API packages can be found in the package summaries:

- `java.security`
- `javax.crypto`
- `java.security.cert`
- `java.security.spec`
- `javax.crypto.spec`
- `java.security.interfaces`
- `javax.crypto.interfaces`

The Provider Class

The term "Cryptographic Service Provider" (used interchangeably with "provider" in this document) refers to a package or set of packages that supply a concrete implementation of a subset of the JDK Security API cryptography features. The `Provider` *class* is the interface to such a package or set of packages. It has methods for accessing the provider name, version number, and other information. Please note that in addition to registering implementations of cryptographic services, the `Provider` class can also be used to register implementations of other security services that might get defined as part of the JDK Security API or one of its extensions.

To supply implementations of cryptographic services, an entity (e.g., a development group) writes the implementation code and creates a subclass of the `Provider` class. The constructor of the `Provider` subclass sets the values of various properties; the JDK Security API uses these values to look up the services that the provider implements. In other words, the subclass specifies the names of the classes implementing the services.

Figure 2-4 Provider Class

Copyright © 1993, 2026, Oracle and/or its affiliates.

There are several types of services that can be implemented by provider packages; See Engine Classes and Algorithms.

The different implementations may have different characteristics. Some may be software-based, while others may be hardware-based. Some may be platform-independent, while others may be platform-specific. Some provider source code may be available for review and evaluation, while some may not. The JCA lets both end-users and developers decide what their needs are.

You can find information about how end-users install the cryptography implementations that fit their needs, and how developers request the implementations that fit theirs.

> **Note**
>
> To implement a provider, see Steps to Implement and Integrate a Provider.

How Provider Implementations Are Requested and Supplied

For each engine class (see Engine Classes and Algorithms) in the API, a implementation instance is requested and instantiated by calling one of the `getInstance` methods on the engine class, specifying the name of the desired algorithm and, optionally, the name of the provider (or the `Provider` class) whose implementation is desired.

```
static EngineClassName getInstance(String algorithm)
    throws NoSuchAlgorithmException

static EngineClassName getInstance(String algorithm, String provider)
    throws NoSuchAlgorithmException, NoSuchProviderException

static EngineClassName getInstance(String algorithm, Provider provider)
    throws NoSuchAlgorithmException
```

where

EngineClassName

is the desired engine type (for example, `Signature`, `MessageDigest`, or `Cipher`). For example:

```
Signature sig = Signature.getInstance("SHA256withRSA");
KeyAgreement ka = KeyAgreement.getInstance("DH");
```

return an instance of the "SHA256withRSA" `Signature` and "DH" `KeyAgreement` objects, respectively.

Java Security Standard Algorithm Names contains the list of names that have been standardized for use with the Java environment. Some providers may choose to also include alias names that also refer to the same algorithm. For example, the "SHA256" algorithm might be referred to as "SHA-256". Applications should use standard names instead of an alias, as not all providers may alias algorithm names in the same way.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The algorithm name is not case-sensitive. For example, all the following calls are equivalent:
>
> ```
> Signature.getInstance("SHA256withRSA")
> Signature.getInstance("sha256withrsa")
> Signature.getInstance("Sha256WithRsa")
> ```

If no provider is specified, `getInstance` searches the registered providers for an implementation of the requested cryptographic service associated with the named algorithm. In any given Java Virtual Machine (JVM), providers are installed in a given *preference order*, the order in which the provider list is searched if a specific provider is not requested. (See Installing Providers.) For example, suppose there are two providers installed in a JVM, `PROVIDER_1` and `PROVIDER_2`. Assume that:

- `PROVIDER_1` implements SHA256withRSA and AES. `PROVIDER_1` has preference order 1 (the highest priority).
- `PROVIDER_2` implements SHA256withRSA, SHA256withDSA, and RC5. `PROVIDER_2` has preference order 2.

Now let's look at three scenarios:

1. We are looking for an SHA256withRSA implementation: Both providers supply such an implementation. The `PROVIDER_1` implementation is returned since `PROVIDER_1` has the highest priority and is searched first.
2. We are looking for an SHA256withDSA implementation: `PROVIDER_1` is first searched for it. No implementation is found, so `PROVIDER_2` is searched. Because an implementation is found, it is returned.
3. We are looking for a SHA256withECDSA implementation: Because no installed provider implements it, a `NoSuchAlgorithmException` is thrown.

The `getInstance` methods that include a provider argument are for developers who want to specify which provider they want an algorithm from. A federal agency, for example, will want to use a provider implementation that has received federal certification. Let's assume that `PROVIDER_1` has not received such certification while `PROVIDER_2` has received it.

A federal agency program would then have the following call, specifying `PROVIDER_2` since it has the certified implementation:

```
Signature s = Signature.getInstance("SHA256withRSA", "PROVIDER_2");
```

In this case, if `PROVIDER_2` was not installed, a `NoSuchProviderException` would be thrown, even if another installed provider implements the algorithm requested.

A program also has the option of getting a list of all the installed providers (using the `getProviders` method in The Security Class class) and choosing one from the list.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> General purpose applications **SHOULD NOT** request cryptographic services from specific providers. Otherwise, applications are tied to specific providers which may not be available on other Java implementations. They also might not be able to take advantage of available optimized providers (for example hardware accelerators via PKCS11 or native OS implementations such as Microsoft's MSCAPI) that have a higher preference order than the specific requested provider.

Installing Providers

In order to be used, a cryptographic provider must first be installed, then registered either statically or dynamically. There are a variety of Sun providers shipped with this release, for instance, `SUN`, `SunJCE`, `SunJSSE`, and `SunRsaSign`, that are already installed and registered. The following sections describe how to install and register additional providers.

All JDK providers are already installed and registered. However, if you require any third-party providers, see Step 8: Prepare for Testing from Steps to Implement and Integrate a Provider for information about how to add providers to the class or module path and register providers, statically or dynamically.

Provider Class Methods

Each `Provider` class instance has a (currently case-sensitive) name, a version number, and a string description of the provider and its services.

You can query the `Provider` instance for this information by calling the following methods:

```
public String getName()
public double getVersion()
public String getInfo()
```

The Security Class

The `Security` class manages installed providers and security-wide properties. It only contains static methods and is never instantiated.

Managing Providers

The following tables summarize the methods in the `Security` class you can use to query which `Providers` are installed, as well as to install or remove providers at runtime.

Querying Providers

Method	Description
`static Provider[] getProviders()`	Returns an array containing all the installed providers (technically, the `Provider` subclass for each package provider). The order of the `Providers` in the array is their preference order.
`static Provider getProvider (String providerName)`	Returns the `Provider` named `providerName`. It returns `null` if the `Provider` is not found.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Adding Providers

Method	Description
`static int addProvider(Provider provider)`	Adds a `Provider` to the end of the list of installed `Providers`. It returns the preference position in which the `Provider` was added, or `-1` if the `Provider` was not added because it was already installed.
`static int insertProviderAt (Provider provider, int position)`	Adds a new `Provider` at a specified position. If the given provider is installed at the requested position, the provider formerly at that position and all providers with a position greater than `position` are shifted up one position (towards the end of the list). This method returns the preference position in which the `Provider` was added, or `-1` if the `Provider` was not added because it was already installed.

Removing Providers

Method	Description
`static void removeProvider(String name)`	Removes the `Provider` with the specified name. It returns silently if the provider is not installed. When the specified provider is removed, all providers located at a position greater than where the specified provider was are shifted down one position (towards the head of the list of installed providers).

> **Note**
>
> If you want to change the preference position of a provider, you must first remove it, and then insert it back in at the new preference position.

Security Properties

The `Security` class maintains a list of system-wide Security Properties. These properties are similar to the `System` properties, but are security-related. These properties can be set statically (through the `<`*`java-home`*`>/conf/security/java.security` file) or dynamically (using an API). See Step 8.1: Configure the Provider from Steps to Implement and Integrate a Provider. for an example of registering a provider statically with the `security.provider.`*`n`* Security Property. If you want to set properties dynamically, programs can use the following methods:

```
static String getProperty(String key)
static void setProperty(String key, String datum)
```

> **Note**
>
> The list of security providers is established during VM startup; therefore, the methods described previously must be used to alter the provider list.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The SecureRandom Class

The SecureRandom class is an engine class (see Engine Classes and Algorithms) that provides cryptographically strong random numbers, either by accessing a pseudo-random number generator (PRNG), a deterministic algorithm that produces a pseudo-random sequence from an initial seed value, or by reading a native source of randomness (for example, `/dev/random` or a true random number generator). One example of a PRNG is the Deterministic Random Bits Generator (DRBG) as specified in NIST SP 800-90Ar1. Other implementations may produce true random numbers, and yet others may use a combination of both techniques. A cryptographically strong random number minimally complies with the statistical random number generator tests specified in FIPS 140-2, Security Requirements for Cryptographic Modules, section 4.9.1.

All Java SE implementations must indicate the strongest (most random) implementation of SecureRandom that they provide in the `securerandom.strongAlgorithms` property of the `java.security.Security` class. This implementation can be used when a particularly strong random value is required.

The `securerandom.drbg.config` property is used to specify the DRBG `SecureRandom` configuration and implementations in the SUN provider. The `securerandom.drbg.config` is a property of the `java.security.Security` class. Other DRBG implementations can also use the `securerandom.drbg.config` property.

Figure 2-5 SecureRandom class

Creating a SecureRandom Object

There are several ways to obtain an instance of `SecureRandom`:

- All Java SE implementations provide a default `SecureRandom` using the no-argument constructor: `new SecureRandom()`. This constructor traverses the list of registered security providers, starting with the most preferred provider, then returns a new `SecureRandom` object from the first provider that supports a `SecureRandom` random number generator (RNG) algorithm. If none of the providers support a RNG algorithm, then it returns a `SecureRandom` object that uses SHA1PRNG from the SUN provider.
- To get a specific implementation of `SecureRandom`, use one of the How Provider Implementations Are Requested and Supplied.
- Use the `getInstanceStrong()` method to obtain a strong `SecureRandom` implementation as defined by the `securerandom.strongAlgorithms` property of the `java.security.Security` class. This property lists platform implementations that are suitable for generating important values.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Seeding or Re-Seeding the SecureRandom Object

The `SecureRandom` object is initialized with a random seed unless the call to `getInstance()` is followed by a call to one of the following `setSeed` methods.

```
void setSeed(byte[] seed)
void setSeed(long seed)
```

You must call `setSeed` before the first `nextBytes` call to prevent any environmental randomness.

The randomness of the bits produced by the `SecureRandom` object depends on the randomness of the seed bits

At any time a `SecureRandom` object may be re-seeded using one of the `setSeed` or `reseed` methods. The given seed for `setSeed` supplements, rather than replaces, the existing seed; therefore, repeated calls are guaranteed never to reduce randomness.

Using a SecureRandom Object

To get random bytes, a caller simply passes an array of any length, which is then filled with random bytes:

```
void nextBytes(byte[] bytes)
```

Generating Seed Bytes

If desired, it is possible to invoke the `generateSeed` method to generate a given number of seed bytes (to seed other random number generators, for example):

```
byte[] generateSeed(int numBytes)
```

The MessageDigest Class

The `MessageDigest` class is an engine class (see Engine Classes and Algorithms) designed to provide the functionality of cryptographically secure message digests such as SHA-256 or SHA-512. A cryptographically secure message digest takes arbitrary-sized input (a byte array), and generates a fixed-size output, called a *digest* or hash.

Figure 2-6 MessageDigest Class

For example, the SHA-256 algorithm produces a 32-byte digest, and SHA-512's is 64 bytes.

A digest has two properties:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- It should be computationally infeasible to find two messages that hash to the same value.
- The digest should not reveal anything about the input that was used to generate it.

Message digests are used to produce unique and reliable identifiers of data. They are sometimes called "checksums" or the "digital fingerprints" of the data. Changes to just one bit of the message should produce a different digest value.

Message digests have many uses and can determine when data has been modified, intentionally or not. When selecting a digest algorithm, one should always consult a recent reference to determine its status and appropriateness for the task at hand.

Creating a MessageDigest Object

Procedure to create a `MessageDigest` object.

- To compute a digest, create a message digest instance. The `MessageDigest` objects are obtained by using one of the `getInstance()` methods in the `MessageDigest` class. See How Provider Implementations Are Requested and Supplied.

  The factory method returns an initialized message digest object. It thus does not need further initialization.

Updating a Message Digest Object

Procedure to update the Message Digest object.

- To calculate the digest of some data, you have to supply the data to the initialized message digest object. It can be provided all at once, or in chunks. Pieces can be fed to the message digest by calling one of the `update` methods:

```
void update(byte input)
void update(byte[] input)
void update(byte[] input, int offset, int len)
```

Computing the Digest

Procedure to compute the digest using different types of `digest()` methods.

The data chunks have to be supplied by calls to `update` method. See Updating a Message Digest Object.

- The digest is computed using a call to one of the `digest` methods:

```
byte[] digest()
byte[] digest(byte[] input)
int digest(byte[] buf, int offset, int len)
```

1. The `byte[] digest()` method return the computed digest.
2. The `byte[] digest(byte[] input)` method does a final `update(input)` with the input byte array before calling `digest()`, which returns the digest byte array.
3. The `int digest(byte[] buf, int offset, int len)` method stores the computed digest in the provided buffer `buf`, starting at `offset`. `len` is the number of bytes in `buf`

Copyright © 1993, 2026, Oracle and/or its affiliates.

allotted for the digest, the method returns the number of bytes actually stored in `buf`. If there is not enough room in the buffer, the method will throw an exception.

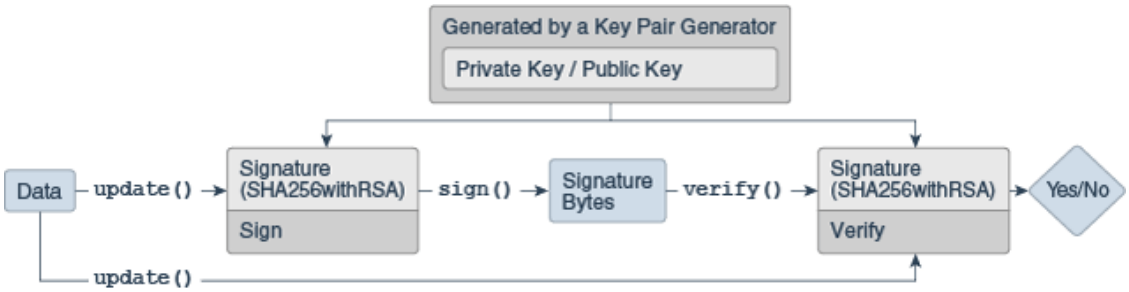
See Computing a MessageDigest Object.

The Signature Class

The `Signature` class is an engine class (see Engine Classes and Algorithms) designed to provide the functionality of a cryptographic digital signature algorithm such as SHA256withDSA or SHA512withRSA. A cryptographically secure signature algorithm takes arbitrary-sized input and a private key and generates a relatively short (often fixed-size) string of bytes, called the *signature*, with the following properties:

- Only the owner of a private/public key pair is able to create a signature. It should be computationally infeasible for anyone having only the public key and a number of signatures to recover the private key.
- Given the public key corresponding to the private key used to generate the signature, it should be possible to verify the authenticity and integrity of the input.

Figure 2-7 Signature Class



A `Signature` object is initialized for signing with a Private Key and is given the data to be signed. The resulting signature bytes are typically kept with the signed data. When verification is needed, another `Signature` object is created and initialized for verification and given the corresponding Public Key. The data and the signature bytes are fed to the signature object, and if the data and signature match, the `Signature` object reports success.

Even though a signature seems similar to a message digest, they have very different purposes in the type of protection they provide. In fact, algorithms such as "SHA256WithRSA" use the message digest "SHA256" to initially "compress" the large data sets into a more manageable form, then sign the resulting 32 byte message digest with the "RSA" algorithm.

For an example for signing and verifying data, see Generating and Verifying a Signature Using Generated Keys.

Signature Object States

`Signature` objects are modal objects. This means that a `Signature` object is always in a given state, where it may only do one type of operation.

States are represented as final integer constants defined in their respective classes.

The three states a `Signature` object may have are:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- UNINITIALIZED
- SIGN
- VERIFY

When it is first created, a Signature object is in the UNINITIALIZED state. The Signature class defines two initialization methods, initSign and initVerify, which change the state to SIGN and VERIFY , respectively.

Creating a Signature Object

The first step for signing or verifying a signature is to create a Signature instance.

Signature objects are obtained by using one of the Signature getInstance() static factory methods. See How Provider Implementations Are Requested and Supplied.

Initializing a Signature Object

A Signature object must be initialized before it is used. The initialization method depends on whether the object is going to be used for signing or for verification.

If it is going to be used for signing, the object must first be initialized with the private key of the entity whose signature is going to be generated. This initialization is done by calling the method:

```
final void initSign(PrivateKey privateKey)
```

This method puts the Signature object in the SIGN state. If instead the Signature object is going to be used for verification, it must first be initialized with the public key of the entity whose signature is going to be verified. This initialization is done by calling either of these methods:

```
final void initVerify(PublicKey publicKey)

final void initVerify(Certificate certificate)
```

This method puts the Signature object in the VERIFY state.

Signing with a Signature Object

If the Signature object has been initialized for signing (if it is in the SIGN state), the data to be signed can then be supplied to the object. This is done by making one or more calls to one of the update methods:

```
final void update(byte b)
final void update(byte[] data)
final void update(byte[] data, int off, int len)
```

Calls to the update method(s) should be made until all the data to be signed has been supplied to the Signature object.

Copyright © 1993, 2026, Oracle and/or its affiliates.

To generate the signature, simply call one of the `sign` methods:

```
final byte[] sign()
final int sign(byte[] outbuf, int offset, int len)
```

The first method returns the signature result in a byte array. The second stores the signature result in the provided buffer *outbuf*, starting at *offset*. *len* is the number of bytes in *outbuf* allotted for the signature. The method returns the number of bytes actually stored.

Signature encoding is algorithm specific. See Java Security Standard Algorithm Names to know more about the use of ASN.1 encoding in the Java Cryptography Architecture.

A call to a `sign` method resets the signature object to the state it was in when previously initialized for signing via a call to `initSign`. That is, the object is reset and available to generate another signature with the same private key, if desired, via new calls to `update` and `sign`.

Alternatively, a new call can be made to `initSign` specifying a different private key, or to `initVerify` (to initialize the `Signature` object to verify a signature).

Verifying with a Signature Object

If the `Signature` object has been initialized for verification (if it is in the `VERIFY` state), it can then verify if an alleged signature is in fact the authentic signature of the data associated with it. To start the process, the data to be verified (as opposed to the signature itself) is supplied to the object. The data is passed to the object by calling one of the `update` methods:

```
final void update(byte b)
final void update(byte[] data)
final void update(byte[] data, int off, int len)
```

Calls to the `update` method(s) should be made until all the data to be verified has been supplied to the `Signature` object. The signature can now be verified by calling one of the `verify` methods:

```
final boolean verify(byte[] signature)

final boolean verify(byte[] signature, int offset, int length)
```

The argument must be a byte array containing the signature. This byte array would hold the signature bytes which were returned by a previous call to one of the `sign` methods.

The `verify` method returns a `boolean` indicating whether or not the encoded signature is the authentic signature of the data supplied to the `update` method(s).

A call to the `verify` method resets the signature object to its state when it was initialized for verification via a call to `initVerify`. That is, the object is reset and available to verify another signature from the identity whose public key was specified in the call to `initVerify`.

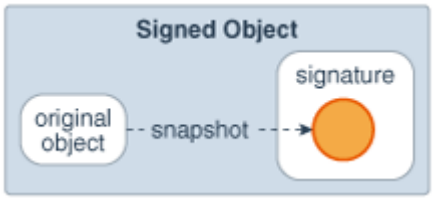
Alternatively, a new call can be made to `initVerify` specifying a different public key (to initialize the `Signature` object for verifying a signature from a different entity), or to `initSign` (to initialize the `Signature` object for generating a signature).

Copyright © 1993, 2026, Oracle and/or its affiliates.

The SignedObject Class

This class is an essential building block for other security primitives. `SignedObject` contains another `Serializable` object, the (to-be-)signed object and its signature. If the signature is not null, it contains a valid digital signature of the signed object. This is illustrated in Figure 2-8.

Figure 2-8 Signed Object Contents



The underlying signing algorithm is set through a Signature object as a parameter to the `sign` method call, and the algorithm can be, among others, the NIST standard DSA, using DSA and SHA-256. The algorithm is specified using the same convention for signatures, such as "SHA/DSA".

The signed object is a "deep copy" (in serialized form) of an original object. Once the copy is made, further manipulation of the original object has no side effect on the copy. A signed object is immutable.

A typical example of creating a signed object is the following:

```
Signature signingEngine = Signature.getInstance(algorithm;
SignedObject so = new SignedObject(myobject, signingKey, signingEngine);
```

A typical example of verification is the following (having received `SignedObject so`), where the first line is not needed if the name of the algorithm is known:

```
String algorithm = so.getAlgorithm();
Signature verificationEngine = Signature.getInstance(algorithm);
so.verify(verificationEngine);
```

Potential applications of `SignedObject` include:

- It can be used internally to any Java application environment as an unforgeable authorization token – one that can be passed around without the fear that the token can be maliciously modified without being detected.
- It can be used to sign and serialize data/object for storage outside the Java runtime (e.g., storing critical access control data on disk).
- Nested `SignedObjects` can be used to construct a logical sequence of signatures, resembling a chain of authorization and delegation.

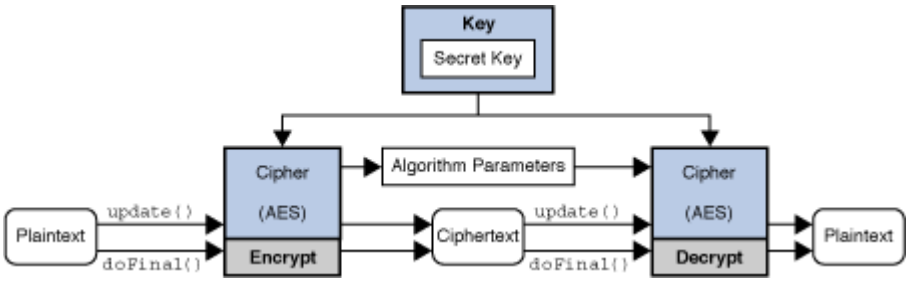
It is intended that this class can be subclassed in the future to allow multiple signatures on the same signed object. In that case, existing method calls in this base class will be fully compatible in semantics. In particular, any `get` method will return the unique value if there is only one signature, and will return an arbitrary one from the set of signatures if there is more than one signature.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Cipher Class

The `Cipher` class provides the functionality of a cryptographic cipher used for encryption and decryption. Encryption is the process of taking data (called *cleartext*) and a *key*, and producing data (*ciphertext*) meaningless to a third-party who does not know the key. Decryption is the inverse process: that of taking ciphertext and a key and producing cleartext.

Figure 2-9 The Cipher Class



Symmetric Versus Asymmetric Cryptography

There are two major types of encryption: *symmetric* (also known as *secret key*), and *asymmetric* (or *public key cryptography*). In symmetric cryptography, the same secret key to both encrypt and decrypt the data. Keeping the key private is critical to keeping the data confidential. On the other hand, asymmetric cryptography uses a public/private key pair to encrypt data. Data encrypted with one key is decrypted with the other. A user first generates a public/private key pair, and then publishes the public key in a trusted database that anyone can access. A user who wishes to communicate securely with that user encrypts the data using the retrieved public key. Only the holder of the private key will be able to decrypt. Keeping the private key confidential is critical to this scheme.

Asymmetric algorithms (such as RSA) are generally much slower than symmetric ones. These algorithms are not designed for efficiently protecting large amounts of data. In practice, asymmetric algorithms are used to exchange smaller secret keys which are used to initialize symmetric algorithms.

Stream versus Block Ciphers

There are two major types of ciphers: *block* and *stream*. Block ciphers process entire blocks at a time, usually many bytes in length. If there is not enough data to make a complete input block, the data must be *padded*: that is, before encryption, dummy bytes must be added to make a multiple of the cipher's block size. These bytes are then stripped off during the decryption phase. The padding can either be done by the application, or by initializing a cipher to use a padding type such as "PKCS5PADDING". In contrast, stream ciphers process incoming data one small unit (typically a byte or even a bit) at a time. This allows for ciphers to process an arbitrary amount of data without padding.

Modes Of Operation

When encrypting using a simple block cipher, two identical blocks of plaintext will always produce an identical block of cipher text. Cryptanalysts trying to break the ciphertext will have an easier job if they note blocks of repeating text. A cipher mode of operation makes the ciphertext less predictable with output block alterations based on block position or the values of other ciphertext blocks. The first block will need an initial value, and this value is called the *initialization vector (IV)*. Since the IV simply alters the data before any encryption, the IV should

Copyright © 1993, 2026, Oracle and/or its affiliates.

be random but does not necessarily need to be kept secret. There are a variety of modes, such as CBC (Cipher Block Chaining), CFB (Cipher Feedback Mode), and OFB (Output Feedback Mode). ECB (Electronic Codebook Mode) is a mode in which there is no influence from block position or other ciphertext blocks. Because ECB ciphertexts are the same if they use the same plaintext/key, this mode is not typically suitable for cryptographic applications and should not be used.

Some algorithms such as AES and RSA allow for keys of different lengths, but others are fixed, such as 3DES. Encryption using a longer key generally implies a stronger resistance to message recovery. As usual, there is a trade off between security and time, so choose the key length appropriately.

Most algorithms use binary keys. Most humans do not have the ability to remember long sequences of binary numbers, even when represented in hexadecimal. Character passwords are much easier to recall. Because character passwords are generally chosen from a small number of characters (for example, [a-zA-Z0-9]), protocols such as "Password-Based Encryption" (PBE) have been defined which take character passwords and generate strong binary keys. In order to make the task of getting from password to key very time-consuming for an attacker (via so-called "rainbow table attacks" or "precomputed dictionary attacks" where common dictionary word->value mappings are precomputed), most PBE implementations will mix in a random number, known as a *salt*, to reduce the usefulness of precomputed tables.

Newer cipher modes such as Authenticated Encryption with Associated Data (AEAD) (for example, Galois/Counter Mode (GCM)) encrypt data and authenticate the resulting message simultaneously. Additional Associated Data (AAD) can be used during the calculation of the resulting AEAD tag (MAC), but this AAD data is not output as ciphertext. (For example, some data might not need to be kept confidential, but should figure into the tag calculation to detect modifications.) The `Cipher.updateAAD()` methods can be used to include AAD in the tag calculations.

Using an AES Cipher with GCM Mode

The AES cipher with GCM is an AEAD cipher which has different usage patterns than the non-AEAD ciphers. Apart from the regular data, it also takes AAD which is optional for encryption/decryption but AAD must be supplied before data for encryption/decryption. In addition, in order to use GCM securely, callers should not re-use key and IV combinations for encryption. This means that the `Cipher` object should be explicitly re-initialized with a different set of parameters every time for each encryption operation.

Example 2-2 Sample Code for Using an AES Cipher with GCM Mode

```
SecretKey myKey = ...
byte[] myAAD = ...
byte[] plainText = ...
int myTLen = ...
byte[] myIv = ...

GCMParameterSpec myParams = new GCMParameterSpec(myTLen, myIv);
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, myKey, myParams);

// AAD is optional, if present, it must be supplied before any update/doFinal
calls.
c.updateAAD(myAAD);  // if AAD is non-null
byte[] cipherText = new byte[c.getOutputSize(plainText.length)];

// conclusion of encryption operation
int actualOutputLen = c.doFinal(plainText, 0, plainText.length, cipherText);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
// To decrypt, same AAD and GCM parameters must be supplied
c.init(Cipher.DECRYPT_MODE, myKey, myParams);
c.updateAAD(myAAD);
byte[] recoveredText = c.doFinal(cipherText, 0, actualOutputLen);

// MUST CHANGE IV VALUE if the same key were to be used again for encryption
byte[] newIv = ...;
myParams = new GCMParameterSpec(myTLen, newIv);
```

Creating a Cipher Object

`Cipher` objects are obtained by using one of the `Cipher` `getInstance()` static factory methods. See How Provider Implementations Are Requested and Supplied. Here, the algorithm name is slightly different than with other engine classes, in that it specifies not just an algorithm name, but a "transformation". A transformation is a string that describes the operation (or set of operations) to be performed on the given input to produce some output. A transformation always includes the name of a cryptographic algorithm (for example, `AES`), and may be followed by a mode and padding scheme.

A transformation is of the form:

- "*algorithm/mode/padding*" or
- "*algorithm*"

For example, the following are valid transformations:

"*AES/CBC/PKCS5Padding*"
"*AES*"

If just a transformation name is specified, the system will determine if there is an implementation of the requested transformation available in the environment, and if there is more than one, returns if there is a preferred one.

If both a transformation name and a package provider are specified, the system will determine if there is an implementation of the requested transformation in the package requested, and throw an exception if there is not.

It is recommended to use a transformation that fully specifies the algorithm, mode, and padding. By not doing so, the provider will use a default. For example, the SunJCE and SunPKCS11 providers use ECB as the default mode, and PKCS5Padding as the default padding for many symmetric ciphers.

This means that in the case of the `SunJCE` provider:

```
Cipher c1 = Cipher.getInstance("AES/ECB/PKCS5Padding");
```

and

```
Cipher c1 = Cipher.getInstance("AES");
```

are equivalent statements.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> ECB mode is the easiest block cipher mode to use and is the default cipher mode. ECB works well for single blocks of data and can be parallelized but generally should not be used for encrypting multiple data blocks due to characteristics of the mode. This could result in trivial and full disclosure of confidential data. While this mode is available for use, it should only be used with an understanding of the cryptographic risks involved.

Using modes such as CFB and OFB, block ciphers can encrypt data in units smaller than the cipher's actual block size. When requesting such a mode, you may optionally specify the number of bits to be processed at a time by appending this number to the mode name as shown in the "*AES/CFB8/NoPadding*" and "*AES/OFB32/PKCS5Padding*" transformations. If no such number is specified, a provider-specific default is used. (For example, the `SunJCE` provider uses a default of 256 bits for AES.) Thus, block ciphers can be turned into byte-oriented stream ciphers by using an 8 bit mode such as CFB8 or OFB8.

Java Security Standard Algorithm Names contains a list of standard names that can be used to specify the algorithm name, mode, and padding scheme components of a transformation.

The objects returned by factory methods are uninitialized, and must be initialized before they become usable.

Initializing a Cipher Object

A `Cipher` object obtained through `getInstance` must be initialized for one of four modes, which are defined as final integer constants in the `Cipher` class. The modes can be referenced by their symbolic names:

ENCRYPT_MODE
Encryption of data.

DECRYPT_MODE
Decryption of data.

WRAP_MODE
Wrapping a `java.security.Key` into bytes so that the key can be securely transported.

UNWRAP_MODE
Unwrapping of a previously wrapped key into a `java.security.Key` object.

Each of the `Cipher` initialization methods takes an operational mode parameter (`opmode`), and initializes the `Cipher` object for that mode. Other parameters include the key (`key`) or certificate containing the key (`certificate`), algorithm parameters (`params`), and a source of randomness (`random`).

To initialize a `Cipher` object, call one of the following `init` methods:

```
public void init(int opmode, Key key);

public void init(int opmode, Certificate certificate);

public void init(int opmode, Key key, SecureRandom random);

public void init(int opmode, Certificate certificate,
                 SecureRandom random);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
public void init(int opmode, Key key,
                 AlgorithmParameterSpec params);

public void init(int opmode, Key key,
                 AlgorithmParameterSpec params, SecureRandom random);

public void init(int opmode, Key key,
                 AlgorithmParameters params);

public void init(int opmode, Key key,
                 AlgorithmParameters params, SecureRandom random);
```

If a `Cipher` object that requires parameters (for example, an initialization vector) is initialized for encryption, and no parameters are supplied to the `init` method, the underlying cipher implementation is supposed to supply the required parameters itself, either by generating random parameters or by using a default, provider-specific set of parameters.

However, if a `Cipher` object that requires parameters is initialized for decryption, and no parameters are supplied to the `init` method, an `InvalidKeyException` or `InvalidAlgorithmParameterException` exception will be raised, depending on the `init` method that has been used.

See Managing Algorithm Parameters.

The same parameters that were used for encryption must be used for decryption.

Note that when a `Cipher` object is initialized, it loses all previously-acquired state. In other words, initializing a `Cipher` is equivalent to creating a new instance of that `Cipher`, and initializing it. For example, if a `Cipher` is first initialized for decryption with a given key, and then initialized for encryption, it will lose any state acquired while in decryption mode.

Encrypting and Decrypting Data

Data can be encrypted or decrypted in one step (*single-part operation*) or in multiple steps (*multiple-part operation*). A multiple-part operation is useful if you do not know in advance how long the data is going to be, or if the data is too long to be stored in memory all at once.

To encrypt or decrypt data in a single step, call one of the `doFinal` methods:

```
public byte[] doFinal(byte[] input);

public byte[] doFinal(byte[] input, int inputOffset, int inputLen);

public int doFinal(byte[] input, int inputOffset,
                   int inputLen, byte[] output);

public int doFinal(byte[] input, int inputOffset,
                   int inputLen, byte[] output, int outputOffset)
```

To encrypt or decrypt data in multiple steps, call one of the `update` methods:

```
public byte[] update(byte[] input);

public byte[] update(byte[] input, int inputOffset, int inputLen);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
public int update(byte[] input, int inputOffset, int inputLen,
                  byte[] output);

public int update(byte[] input, int inputOffset, int inputLen,
                  byte[] output, int outputOffset)
```

A multiple-part operation must be terminated by one of the these `doFinal` methods (if there is still some input data left for the last step), or by one of the following `doFinal` methods (if there is no input data left for the last step):

```
public byte[] doFinal();

public int doFinal(byte[] output, int outputOffset);
```

All the `doFinal` methods take care of any necessary padding (or unpadding), if padding (or unpadding) has been requested as part of the specified transformation.

A call to `doFinal` resets the `Cipher` object to the state it was in when initialized via a call to `init`. That is, the `Cipher` object is reset and available to encrypt or decrypt (depending on the operation mode that was specified in the call to `init`) more data.

Wrapping and Unwrapping Keys

Wrapping a key enables secure transfer of the key from one place to another.

The `wrap` and `unwrap` methods make it more convenient to write code since they work with key objects directly. These methods also enable the possibility of secure transfer of hardware-based keys.

To **wrap** a `Key`, first initialize the `Cipher` object for WRAP_MODE, and then call the following:

```
public final byte[] wrap(Key key);
```

If you are supplying the wrapped key bytes (the result of calling `wrap`) to someone else who will unwrap them, be sure to also send additional information the recipient will need in order to do the `unwrap`:

- The name of the key algorithm.
- The type of the wrapped key (one of `Cipher.SECRET_KEY`, `Cipher.PRIVATE_KEY`, or `Cipher.PUBLIC_KEY`).

The key algorithm name can be determined by calling the `getAlgorithm` method from the Key interface:

```
public String getAlgorithm();
```

To **unwrap** the bytes returned by a previous call to `wrap`, first initialize a `Cipher` object for UNWRAP_MODE, then call the following:

```
public final Key unwrap(byte[] wrappedKey,
                        String wrappedKeyAlgorithm,
                        int wrappedKeyType));
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Here, `wrappedKey` is the bytes returned from the previous call to wrap, `wrappedKeyAlgorithm` is the algorithm associated with the wrapped key, and `wrappedKeyType` is the type of the wrapped key. This must be one of `Cipher.SECRET_KEY`, `Cipher.PRIVATE_KEY`, or `Cipher.PUBLIC_KEY`.

Managing Algorithm Parameters

The parameters being used by the underlying `Cipher` implementation, which were either explicitly passed to the `init` method by the application or generated by the underlying implementation itself, can be retrieved from the `Cipher` object by calling its `getParameters` method, which returns the parameters as a `java.security.AlgorithmParameters` object (or `null` if no parameters are being used). If the parameter is an initialization vector (IV), it can also be retrieved by calling the `getIV` method.

In the following example, a `Cipher` object implementing password-based encryption (PBE) is initialized with just a key and no parameters. However, the selected algorithm for password-based encryption requires two parameters - a *salt* and an *iteration count*. Those will be generated by the underlying algorithm implementation itself. The application can retrieve the generated parameters from the `Cipher` object, see Example 2-3.

The same parameters that were used for encryption must be used for decryption. They can be instantiated from their encoding and used to initialize the corresponding `Cipher` object for decryption, see Example 2-4.

If you did not specify any parameters when you initialized a `Cipher` object, and you are not sure whether or not the underlying implementation uses any parameters, you can find out by simply calling the `getParameters` method of your `Cipher` object and checking the value returned. A return value of `null` indicates that no parameters were used.

The following cipher algorithms implemented by the `SunJCE` provider use parameters:

- AES, DES-EDE, and Blowfish, when used in feedback (i.e., CBC, CFB, OFB, or PCBC) mode, use an initialization vector (IV). The `javax.crypto.spec.IvParameterSpec` class can be used to initialize a `Cipher` object with a given IV. In addition, CTR and GCM modes require an IV.
- PBE `Cipher` algorithms use a set of parameters, comprising a salt and an iteration count. The `javax.crypto.spec.PBEParameterSpec` class can be used to initialize a `Cipher` object implementing a PBE algorithm (for example: PBEWithHmacSHA256AndAES_256) with a given salt and iteration count.

Note that you do not have to worry about storing or transferring any algorithm parameters for use by the decryption operation if you use the `SealedObject` class. This class attaches the parameters used for sealing (encryption) to the encrypted object contents, and uses the same parameters for unsealing (decryption).

Example 2-3 Sample Code for Retrieving Parameters from the Cipher Object

The application can retrieve the generated parameters for encryption from the `Cipher` object as follows:

```
// get cipher object for password-based encryption
Cipher c = Cipher.getInstance("PBEWithHmacSHA256AndAES_256");

// initialize cipher for encryption, without supplying
// any parameters. Here, "myKey" is assumed to refer
// to an already-generated key.
c.init(Cipher.ENCRYPT_MODE, myKey);

// encrypt some data and store away ciphertext
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
// for later decryption
byte[] cipherText = c.doFinal("This is just an example".getBytes());

// retrieve parameters generated by underlying cipher
// implementation
AlgorithmParameters algParams = c.getParameters();

// get parameter encoding and store it away
byte[] encodedAlgParams = algParams.getEncoded();
```

Example 2-4 Sample Code for Initializing the Cipher Object for Decryption

The same parameters that were used for encryption must be used for decryption. They can be instantiated from their encoding and used to initialize the corresponding `Cipher` object for decryption as follows:

```
// get parameter object for password-based encryption
AlgorithmParameters algParams;
algParams = AlgorithmParameters.getInstance("PBEWithHmacSHA256AndAES_256");

// initialize with parameter encoding from the previous example
algParams.init(encodedAlgParams);

// get cipher object for password-based encryption
Cipher c = Cipher.getInstance("PBEWithHmacSHA256AndAES_256");

// initialize cipher for decryption, using one of the
// init() methods that takes an AlgorithmParameters
// object, and pass it the algParams object from the previous example
c.init(Cipher.DECRYPT_MODE, myKey, algParams);
```

Cipher Output Considerations

Some of the `update` and `doFinal` methods of `Cipher` allow the caller to specify the output buffer into which to encrypt or decrypt the data. In these cases, it is important to pass a buffer that is large enough to hold the result of the encryption or decryption operation.

The following method in `Cipher` can be used to determine how big the output buffer should be:

```
public int getOutputSize(int inputLen)
```

Other Cipher-based Classes

There are some helper classes which internally use `Ciphers` to provide easy access to common cipher uses.

Topics

The Cipher Stream Classes

The SealedObject Class

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Cipher Stream Classes

The `CipherInputStream` and `CipherOutputStream` classes are Cipher stream classes.

The CipherInputStream Class

This class is a `FilterInputStream` that encrypts or decrypts the data passing through it. It is composed of an `InputStream`. `CipherInputStream` represents a secure input stream into which a `Cipher` object has been interposed. The `read` methods of `CipherInputStream` return data that are read from the underlying `InputStream` but have additionally been processed by the embedded `Cipher` object. The `Cipher` object must be fully initialized before being used by a CipherInputStream.

For example, if the embedded `Cipher` has been initialized for decryption, the `CipherInputStream` will attempt to decrypt the data it reads from the underlying `InputStream` before returning them to the application.

This class adheres strictly to the semantics, especially the failure semantics, of its ancestor classes `java.io.FilterInputStream` and `java.io.InputStream`. This class has exactly those methods specified in its ancestor classes, and overrides them all, so that the data are additionally processed by the embedded cipher. Moreover, this class catches all exceptions that are not thrown by its ancestor classes. In particular, the `skip(long)` method skips only data that has been processed by the `Cipher`.

It is crucial for a programmer using this class not to use methods that are not defined or overridden in this class (such as a new method or constructor that is later added to one of the super classes), because the design and implementation of those methods are unlikely to have considered security impact with regard to `CipherInputStream`. See Example 2-5 for its usage, suppose `cipher1` has been initialized for encryption. The program reads and encrypts the content from the file `/tmp/a.txt` and then stores the result (the encrypted bytes) in `/tmp/b.txt`.

Example 2-6 demonstrates how to easily connect several instances of `CipherInputStream` and `FileInputStream`. In this example, assume that `cipher1` and `cipher2` have been initialized for encryption and decryption (with corresponding keys), respectively. The program copies the content from file `/tmp/a.txt` to `/tmp/b.txt`, except that the content is first encrypted and then decrypted back when it is read from `/tmp/a.txt`. Of course since this program simply encrypts text and decrypts it back right away, it's actually not very useful except as a simple way of illustrating chaining of `CipherInputStreams`.

Note that the read methods of the `CipherInputStream` will block until data is returned from the underlying cipher. If a block cipher is used, a full block of cipher text will have to be obtained from the underlying `InputStream`.

Example 2-5 Sample Code for Using CipherInputStream and FileInputStream

The following code demonstrates how to use a `CipherInputStream` containing that cipher and a `FileInputStream` in order to encrypt input stream data:

```
try (FileInputStream fis = new FileInputStream("/tmp/a.txt");
        CipherInputStream cis = new CipherInputStream(fis, cipher1);
        FileOutputStream fos = new FileOutputStream("/tmp/b.txt")) {
    byte[] b = new byte[8];
    int i = cis.read(b);
    while (i != -1) {
        fos.write(b, 0, i);
        i = cis.read(b);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    }
}
```

Example 2-6 Sample Code for Connecting CipherInputStream and FileInputStream

The following example demonstrates how to easily connect several instances of `CipherInputStream` and `FileInputStream`. In this example, assume that `cipher1` and `cipher2` have been initialized for encryption and decryption (with corresponding keys), respectively:

```
try (FileInputStream fis = new FileInputStream("/tmp/a.txt");
        CipherInputStream cis1 = new CipherInputStream(fis, cipher1);
        CipherInputStream cis2 = new CipherInputStream(cis1, cipher2);
        FileOutputStream fos = new FileOutputStream("/tmp/b.txt")) {
    byte[] b = new byte[8];
    int i = cis2.read(b);
    while (i != -1) {
        fos.write(b, 0, i);
        i = cis2.read(b);
    }
}
```

The CipherOutputStream Class

This class is a `FilterOutputStream` that encrypts or decrypts the data passing through it. It is composed of an `OutputStream`, or one of its subclasses, and a `Cipher`. `CipherOutputStream` represents a secure output stream into which a `Cipher` object has been interposed. The `write` methods of `CipherOutputStream` first process the data with the embedded `Cipher` object before writing them out to the underlying `OutputStream`. The `Cipher` object must be fully initialized before being used by a `CipherOutputStream`.

For example, if the embedded `Cipher` has been initialized for encryption, the `CipherOutputStream` will encrypt its data, before writing them out to the underlying output stream.

This class adheres strictly to the semantics, especially the failure semantics, of its ancestor classes `java.io.OutputStream` and `java.io.FilterOutputStream`. This class has exactly those methods specified in its ancestor classes, and overrides them all, so that all data are additionally processed by the embedded cipher. Moreover, this class catches all exceptions that are not thrown by its ancestor classes.

It is crucial for a programmer using this class not to use methods that are not defined or overridden in this class (such as a new method or constructor that is later added to one of the super classes), because the design and implementation of those methods are unlikely to have considered security impact with regard to `CipherOutputStream`.

See Example 2-7 , for its usage, suppose `cipher1` has been initialized for encryption. The program reads the content from the file `/tmp/a.txt`, then encrypts and stores the result (the encrypted bytes) in `/tmp/b.txt`.

Example 2-7 demonstrates how to easily connect several instances of `CipherOutputStream` and `FileOutputStream`. In this example, assume that `cipher1` and `cipher2` have been initialized for decryption and encryption (with corresponding keys), respectively. The program copies the content from file `/tmp/a.txt` to `/tmp/b.txt`, except that the content is first encrypted and then decrypted back before it is written to `/tmp/b.txt`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

One thing to keep in mind when using *block* cipher algorithms is that a full block of plaintext data must be given to the `CipherOutputStream` before the data will be encrypted and sent to the underlying output stream.

There is one other important difference between the `flush` and `close` methods of this class, which becomes even more relevant if the encapsulated `Cipher` object implements a block cipher algorithm with padding turned on:

- `flush` flushes the underlying `OutputStream` by forcing any buffered output bytes that have already been processed by the encapsulated Cipher object to be written out. Any bytes buffered by the encapsulated `Cipher` object and waiting to be processed by it will **not** be written out.
- `close` closes the underlying `OutputStream` and releases any system resources associated with it. It invokes the `doFinal` method of the encapsulated `Cipher` object, causing any bytes buffered by it to be processed and written out to the underlying stream by calling its `flush` method.

Example 2-7 Sample Code for Using CipherOutputStream and FileOutputStream

```
CipherOutputStreamFileOutputStream

try (FileInputStream fis = new FileInputStream("/tmp/a.txt");
        FileOutputStream fos = new FileOutputStream("/tmp/b.txt");
        CipherOutputStream cos = new CipherOutputStream(fos, cipher1)) {
    byte[] b = new byte[8];
    int i = fis.read(b);
    while (i != -1) {
        cos.write(b, 0, i);
        i = fis.read(b);
    }
    cos.flush();
}
```

Example 2-8 Sample Code for Connecting CipherOutputStream and FileOutputStream

```
CipherOutputStreamFileOutputStreamcipher1cipher2

try (FileInputStream fis = new FileInputStream("/tmp/a.txt");
        FileOutputStream fos = new FileOutputStream("/tmp/b.txt");
        CipherOutputStream cos1 = new CipherOutputStream(fos, cipher1);
        CipherOutputStream cos2 = new CipherOutputStream(cos1, cipher2)) {
    byte[] b = new byte[8];
    int i = fis.read(b);
    while (i != -1) {
        cos2.write(b, 0, i);
        i = fis.read(b);
    }
    cos2.flush();
}
```

The SealedObject Class

This class enables a programmer to create an object and protect its confidentiality with a cryptographic algorithm.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Given any object that implements the `java.io.Serializable` interface, one can create a `SealedObject` that encapsulates the original object, in serialized format (i.e., a "deep copy"), and seals (encrypts) its serialized contents, using a cryptographic algorithm such as AES, to protect its confidentiality. The encrypted content can later be decrypted (with the corresponding algorithm using the correct decryption key) and de-serialized, yielding the original object.

A typical usage is illustrated in the following code segment: In order to seal an object, you create a `SealedObject` from the object to be sealed and a fully initialized `Cipher` object that will encrypt the serialized object contents. In this example, the String "This is a secret" is sealed using the AES algorithm. Note that any algorithm parameters that may be used in the sealing operation are stored inside of `SealedObject`:

```
// create Cipher object
// NOTE: sKey is assumed to refer to an already-generated
// secret AES key.
Cipher c = Cipher.getInstance("AES");
c.init(Cipher.ENCRYPT_MODE, sKey);

// do the sealing
SealedObject so = new SealedObject("This is a secret", c);
```

The original object that was sealed can be recovered in two different ways:

- by using a `Cipher` object that has been initialized with the exact same algorithm, key, padding scheme, etc., that were used to seal the object:

  ```
  c.init(Cipher.DECRYPT_MODE, sKey);
  try {
      String s = (String)so.getObject(c);
  } catch (Exception e) {
      // do something
  };
  ```

  This approach has the advantage that the party who unseals the sealed object does not require knowledge of the decryption key. For example, after one party has initialized the cipher object with the required decryption key, it could hand over the cipher object to another party who then unseals the sealed object.

- by using the appropriate decryption key (since AES is a symmetric encryption algorithm, we use the same key for sealing and unsealing):

  ```
  try {
      String s = (String)so.getObject(sKey);
  } catch (Exception e) {
      // do something
  };
  ```

  In this approach, the `getObject` method creates a cipher object for the appropriate decryption algorithm and initializes it with the given decryption key and the algorithm parameters (if any) that were stored in the sealed object. This approach has the advantage that the party who unseals the object does not need to keep track of the parameters (e.g., the IV) that were used to seal the object.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The KEM Class

The `KEM` class is an engine class (see Engine Classes and Algorithms) that provides the functionality of a Key Encapsulation Mechanism (KEM).

You can use the KEM to secure symmetric keys using asymmetric or public key cryptography between two parties. The sender calls the `encapsulate` method to generate a secret key and a key encapsulation message, and the receiver calls the `decapsulate` method to recover the same secret key from the key encapsulation message.

Preparation

The receiver needs to create a key pair using a `KeyPairGenerator`. The public key is published and made avaiable to the sender, and the private key is kept in secret.

Creating KEM Objects

Each party needs to create a `KEM` object. `KEM` objects are created by using one of the `KEM` `getInstance()` static factory methods. See How Provider Implementations Are Requested and Supplied.

Creating an Encapsulator and a Decapsulator

On the sender side, call one of the `newEncapsulator` methods of the `KEM` object to create an `encapsulator`. The receiver's public key is used in the process. On the receiver side, call one of the `newDecapsulator` methods of the `KEM` object to create a decapsulator. The receiver's private key is used in the process.

Encapsulation and Decapsulation

The sender calls one of the `encapsulate` methods in the newly created `KEM.Encapsulator` object, which returns a `KEM.Encapsulated` object. The secret key inside the `KEM.Encapsulated` object is kept in secret, and the key encapsulation message inside it is sent to the receiver.

The receiver passes the key encapsulation message from the sender to one of the `decapsulate` methods in the newly created `KEM.Decapsulator` object, which returns a `SecretKey` object. This secret key is identical to the secret key on the sender's side.

The sender can use the key for future secure communications with the receiver.

See Encapsulating and Decapsulating Keys for a code example.

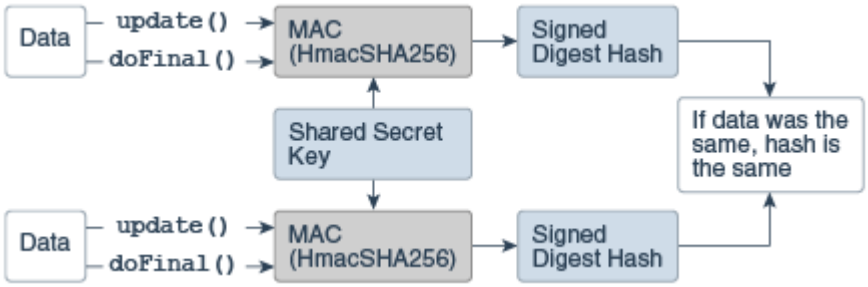
The Mac Class

Similar to a `MessageDigest`, a Message Authentication Code (MAC) provides a way to check the integrity of information transmitted over or stored in an unreliable medium, but includes a secret key in the calculation.

Only someone with the proper key will be able to verify the received message. Typically, message authentication codes are used between two parties that share a secret key in order to validate information transmitted between these parties.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 2-10 The Mac Class



A MAC mechanism that is based on cryptographic hash functions is referred to as HMAC. HMAC can be used with any cryptographic hash function, e.g., SHA-256, in combination with a secret shared key.

The `Mac` class provides the functionality of a Message Authentication Code (MAC). See HMAC-SHA256 Example.

Creating a Mac Object

`Mac` objects are obtained by using one of the `Mac` `getInstance()` static factory methods. See How Provider Implementations Are Requested and Supplied.

Initializing a Mac Object

A Mac object is always initialized with a (secret) key and may optionally be initialized with a set of parameters, depending on the underlying MAC algorithm.

To initialize a Mac object, call one of its `init` methods:

```
public void init(Key key);

public void init(Key key, AlgorithmParameterSpec params);
```

You can initialize your Mac object with any (secret-)key object that implements the `javax.crypto.SecretKey` interface. This could be an object returned by `javax.crypto.KeyGenerator.generateKey()`, or one that is the result of a key agreement protocol, as returned by `javax.crypto.KeyAgreement.generateSecret()`, or an instance of `javax.crypto.spec.SecretKeySpec`.

With some MAC algorithms, the (secret-)key algorithm associated with the (secret-)key object used to initialize the Mac object does not matter (this is the case with the HMAC-MD5 and HMAC-SHA1 implementations of the `SunJCE` provider). With others, however, the (secret-)key algorithm does matter, and an `InvalidKeyException` is thrown if a (secret-)key object with an inappropriate (secret-)key algorithm is used.

Computing a MAC

A MAC can be computed in one step (*single-part operation*) or in multiple steps (*multiple-part operation*). A multiple-part operation is useful if you do not know in advance how long the data is going to be, or if the data is too long to be stored in memory all at once.

Copyright © 1993, 2026, Oracle and/or its affiliates.

To compute the MAC of some data in a single step, call the following `doFinal` method:

```
public byte[] doFinal(byte[] input);
```

To compute the MAC of some data in multiple steps, call one of the `update` methods:

```
public void update(byte input);

public void update(byte[] input);

public void update(byte[] input, int inputOffset, int inputLen);
```

A multiple-part operation must be terminated by the `doFinal` method (if there is still some input data left for the last step), or by one of the following `doFinal` methods (if there is no input data left for the last step):

```
public byte[] doFinal();

public void doFinal(byte[] output, int outOffset);
```

Key Interfaces

The `java.security.Key` interface is the top-level interface for all opaque keys. It defines the functionality shared by all opaque key objects.

To this point, we have focused the high-level uses of the JCA without getting lost in the details of what keys are and how they are generated/represented. It is now time to turn our attention to keys.

An *opaque* key representation is one in which you have no direct access to the key material that constitutes a key. In other words: "opaque" gives you limited access to the key—just the three methods defined by the `Key` interface: `getAlgorithm`, `getFormat`, and `getEncoded`.

This is in contrast to a *transparent* representation, in which you can access each key material value individually, through one of the `get` methods defined in the corresponding `KeySpec` interface (see The KeySpec Interface).

All opaque keys have three characteristics:

An Algorithm
The key algorithm for that key. The key algorithm is usually an encryption or asymmetric operation algorithm (such as `AES`, `DSA` or `RSA`), which will work with those algorithms and with related algorithms (such as `SHA256withRSA`). The name of the algorithm of a key is obtained using this method:

```
String getAlgorithm()
```

An Encoded Form
The external encoded form for the key used when a standard representation of the key is needed outside the Java Virtual Machine, as when transmitting the key to some other party.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The key is encoded according to a standard format (such as X.509 or PKCS8), and is returned using the method:

```
byte[] getEncoded()
```

A Format
The name of the format of the encoded key. It is returned by the method:

```
String getFormat()
```

Keys are generally obtained through key generators such as the `KeyGenerator` class and the `KeyPairGenerator` class, certificates, key specifications (see the The KeySpec Interface) using a `KeyFactory`, or a `Keystore` implementation accessing a keystore database used to manage keys. It is possible to parse encoded keys, in an algorithm-dependent manner, using a `KeyFactory`.

It is also possible to parse certificates, using a `CertificateFactory`.

Here is a list of interfaces which extend the `Key` interface in the `java.security.interfaces` and `javax.crypto.interfaces` packages:

- `SecretKey`
  - `PBEKey`
- `AsymmetricKey`
  - `PrivateKey`
    - `DHPrivateKey`
    - `DSAPrivateKey`
    - `ECPrivateKey`
    - `EdECPrivateKey`
    - `RSAMultiPrimePrivateCrtKey`
    - `RSAPrivateCrtKey`
    - `RSAPrivateKey`
    - `XECPrivateKey`
  - `PublicKey`
    - `DHPublicKey`
    - `DSAPublicKey`
    - `ECPublicKey`
    - `EdECPublicKey`
    - `RSAPublicKey`
    - `XECPublicKey`

Copyright © 1993, 2026, Oracle and/or its affiliates.

The AsymmetricKey Interface

The `AsymmetricKey` interface contains methods that are common to either a public key or a private key.

The PublicKey and PrivateKey Interfaces

The `PublicKey` and `PrivateKey` interfaces (which both extend the `Key` interface) are methodless interfaces, used for type-safety and type-identification.

The KeyPair Class

The `KeyPair` class is a simple holder for a key pair (a public key and a private key).

It has two public methods, one for returning the private key, and the other for returning the public key:

```
PrivateKey getPrivate()
PublicKey getPublic()
```

Key Specification Interfaces and Classes

`Key` objects and key specifications (`KeySpecs`) are two different representations of key data. `Ciphers` use `Key` objects to initialize their encryption algorithms, but keys may need to be converted into a more portable format for transmission or storage.

A *transparent* representation of keys means that you can access each key material value individually, through one of the `get` methods defined in the corresponding specification class. For example, `DSAPrivateKeySpec` defines `getX`, `getP`, `getQ`, and `getG` methods, to access the private key `x`, and the DSA algorithm parameters used to calculate the key: the prime `p`, the sub-prime `q`, and the base `g`. If the key is stored on a hardware device, its specification may contain information that helps identify the key on the device.

This representation is contrasted with an *opaque* representation, as defined by the Key Interfaces interface, in which you have no direct access to the key material fields. In other words, an "opaque" representation gives you limited access to the key—just the three methods defined by the `Key` interface: `getAlgorithm`, `getFormat`, and `getEncoded`.

A key may be specified in an algorithm-specific way, or in an algorithm-independent encoding format (such as ASN.1). For example, a DSA private key may be specified by its components `x`, `p`, `q`, and `g` (see `DSAPrivateKeySpec`), or it may be specified using its DER encoding (see `PKCS8EncodedKeySpec`).

The `KeyFactory` and `SecretKeyFactory` classes can be used to convert between opaque and transparent key representations (that is, between `Keys` and `KeySpecs`, assuming that the operation is possible. (For example, private keys on smart cards might not be able leave the card. Such `Keys` are not convertible.)

In the following sections, we discuss the key specification interfaces and classes in the `java.security.spec` package.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The KeySpec Interface

This interface contains no methods or constants. Its only purpose is to group and provide type safety for all key specifications. All key specifications must implement this interface.

The KeySpec Subinterfaces

Like the `Key` interface, there are a similar set of `KeySpec` interfaces.

- `SecretKeySpec`
- `EncodedKeySpec`
  - `PKCS8EncodedKeySpec`
  - `X509EncodedKeySpec`
- `DESKeySpec`
- `DESedeKeySpec`
- `PBEKeySpec`
- `DHPrivateKeySpec`
- `DSAPrivateKeySpec`
- `ECPrivateKeySpec`
- `RSAPrivateKeySpec`
  - `RSAMultiPrimePrivateCrtKeySpec`
  - `RSAPrivateCrtKeySpec`
- `DHPublicKeySpec`
- `DSAPublicKeySpec`
- `ECPublicKeySpec`
- `RSAPublicKeySpec`

The EncodedKeySpec Class

This abstract class (which implements the The KeySpec Interface interface) represents a public or private key in encoded format. Its `getEncoded` method returns the encoded key:

```
abstract byte[] getEncoded();
```

and its `getFormat` method returns the name of the encoding format:

```
abstract String getFormat();
```

See the next sections for the concrete implementations `PKCS8EncodedKeySpec` and `X509EncodedKeySpec`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The PKCS8EncodedKeySpec Class

This class, which is a subclass of `EncodedKeySpec`, represents the DER encoding of a private key, according to the format specified in the PKCS8 standard.

Its `getEncoded` method returns the key bytes, encoded according to the PKCS8 standard. Its `getFormat` method returns the string "PKCS#8".

The X509EncodedKeySpec Class

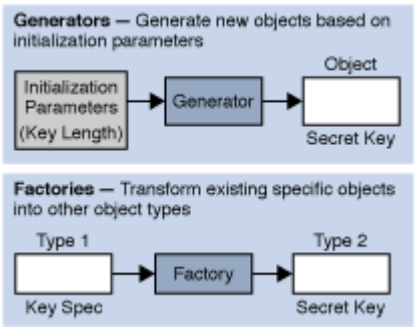
This class, which is a subclass of `EncodedKeySpec`, represents the DER encoding of a public key, according to the format specified in the X.509 standard.

Its `getEncoded` method returns the key bytes, encoded according to the X.509 standard. Its `getFormat` method returns the string "X.509".

Generators and Factories

Newcomers to Java and the JCA APIs in particular sometimes do not grasp the distinction between generators and factories.

Figure 2-11 Generators and Factories



Generators are used to **generate brand new objects**. Generators can be initialized in either an algorithm-dependent or algorithm-independent way. For example, to create a Diffie-Hellman (DH) keypair, an application could specify the necessary P and G values, or the generator could simply be initialized with the appropriate key length, and the generator will select appropriate P and G values. In both cases, the generator will produce brand new keys based on the parameters.

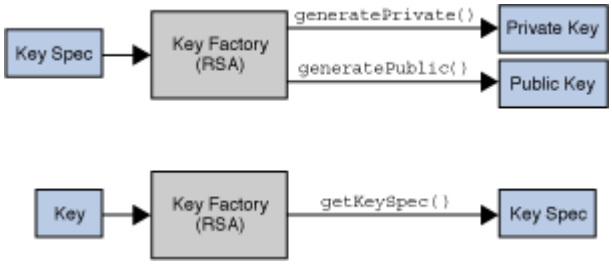
On the other hand, factories are used to **convert data from one existing object type to another**. For example, an application might have available the components of a DH private key and can package them as a The KeySpec Interface, but needs to convert them into a PrivateKey object that can be used by a `KeyAgreement` object, or vice-versa. Or they might have the byte array of a certificate, but need to use a `CertificateFactory` to convert it into a `X509Certificate` object. Applications use factory objects to do the conversion.

The KeyFactory Class

The `KeyFactory` class is an Engine Classes and Algorithms designed to perform conversions between opaque cryptographic Key Interfaces and Key Specification Interfaces and Classes (transparent representations of the underlying key material).

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 2-12 KeyFactory Class



Key factories are bi-directional. They allow you to build an opaque key object from a given key specification (key material), or to retrieve the underlying key material of a key object in a suitable format.

Multiple compatible key specifications can exist for the same key. For example, a DSA public key may be specified by its components `y`, `p`, `q`, and `g` (see `java.security.spec.DSAPublicKeySpec`), or it may be specified using its DER encoding according to the X.509 standard (see The X509EncodedKeySpec Class).

A key factory can be used to translate between compatible key specifications. Key parsing can be achieved through translation between compatible key specifications, e.g., when you translate from `X509EncodedKeySpec` to `DSAPublicKeySpec`, you basically parse the encoded key into its components. For an example, see the end of the Generating/Verifying Signatures Using Key Specifications and KeyFactory section.

Creating a KeyFactory Object

`KeyFactory` objects are obtained by using one of the `KeyFactory`getInstance() static factory methods. See How Provider Implementations Are Requested and Supplied.

Converting Between a Key Specification and a Key Object

If you have a key specification for a public key, you can obtain an opaque `PublicKey` object from the specification by using the `generatePublic` method:

```
PublicKey generatePublic(KeySpec keySpec)
```

Similarly, if you have a key specification for a private key, you can obtain an opaque `PrivateKey` object from the specification by using the `generatePrivate` method:

```
PrivateKey generatePrivate(KeySpec keySpec)
```

Converting Between a Key Object and a Key Specification

If you have a `Key` object, you can get a corresponding key specification object by calling the `getKeySpec` method:

```
KeySpec getKeySpec(Key key, Class keySpec)
```

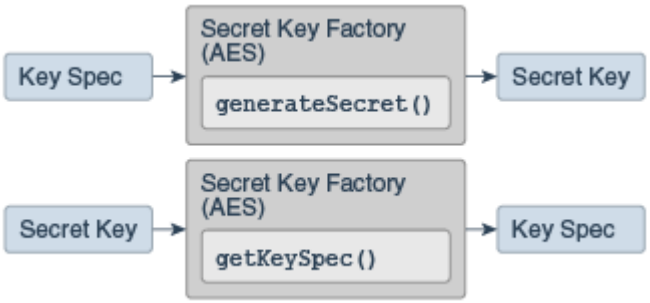

Copyright © 1993, 2026, Oracle and/or its affiliates.

`keySpec` identifies the specification class in which the key material should be returned. It could, for example, be `DSAPublicKeySpec.class` , to indicate that the key material should be returned in an instance of the `DSAPublicKeySpec` class. See Generating/Verifying Signatures Using Key Specifications and KeyFactory.

The SecretKeyFactory Class

The `SecretKeyFactory` class represents a factory for secret keys. Unlike the `KeyFactory` class (see The KeyFactory Class), a `javax.crypto.SecretKeyFactory` object operates only on secret (symmetric) keys, whereas a `java.security.KeyFactory` object processes the public and private key components of a key pair.

Figure 2-13 SecretKeyFactory Class



Key factories are used to convert Key Interfaces (opaque cryptographic keys of type `java.security.Key`) into Key Specification Interfaces and Classes (transparent representations of the underlying key material in a suitable format), and vice versa.

Objects of type `java.security.Key`, of which `java.security.PublicKey`, `java.security.PrivateKey`, and `javax.crypto.SecretKey` are subclasses, are opaque key objects, because you cannot tell how they are implemented. The underlying implementation is provider-dependent, and may be software or hardware based. Key factories allow providers to supply their own implementations of cryptographic keys.

For example, if you have a key specification for a Diffie-Hellman public key, consisting of the public value `y`, the prime modulus `p`, and the base `g`, and you feed the same specification to Diffie-Hellman key factories from different providers, the resulting `PublicKey` objects will most likely have different underlying implementations.

A provider should document the key specifications supported by its secret key factory. For example, the `SecretKeyFactory` for DES keys supplied by the `SunJCE` provider supports `DESKeySpec` as a transparent representation of DES keys, the `SecretKeyFactory` for DES-EDE keys supports `DESedeKeySpec` as a transparent representation of DES-EDE keys, and the `SecretKeyFactory` for PBE supports `PBEKeySpec` as a transparent representation of the underlying password.

The following is an example of how to use a `SecretKeyFactory` to convert secret key data into a `SecretKey` object, which can be used for a subsequent `Cipher` operation:

```
// Note the following bytes are not realistic secret key data
// bytes but are simply supplied as an illustration of using data
// bytes (key material) you already have to build a DESedeKeySpec.

byte[] desEdeKeyData = getKeyData();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
DESedeKeySpec desEdeKeySpec = new DESedeKeySpec(desEdeKeyData);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DESede");
SecretKey secretKey = keyFactory.generateSecret(desEdeKeySpec);
```

In this case, the underlying implementation of `SecretKey` is based on the provider of `KeyFactory`.

An alternative, provider-independent way of creating a functionally equivalent `SecretKey` object from the same key material is to use the `javax.crypto.spec.SecretKeySpec` class, which implements the `javax.crypto.SecretKey` interface:

```
byte[] aesKeyData = getKeyData();
SecretKeySpec secretKey = new SecretKeySpec(aesKeyData, "AES");
```

Creating a SecretKeyFactory Object

`SecretKeyFactory` objects are obtained by using one of the `SecretKeyFactory` `getInstance()` static factory methods. See How Provider Implementations Are Requested and Supplied.

Converting Between a Key Specification and a Secret Key Object

If you have a key specification for a secret key, you can obtain an opaque `SecretKey` object from the specification by using the `generateSecret` method:

```
SecretKey generateSecret(KeySpec keySpec)
```

Converting Between a Secret Key Object and a Key Specification

If you have a `SecretKey` object, you can get a corresponding key specification object by calling the `getKeySpec` method:

```
KeySpec getKeySpec(Key key, Class keySpec)
```

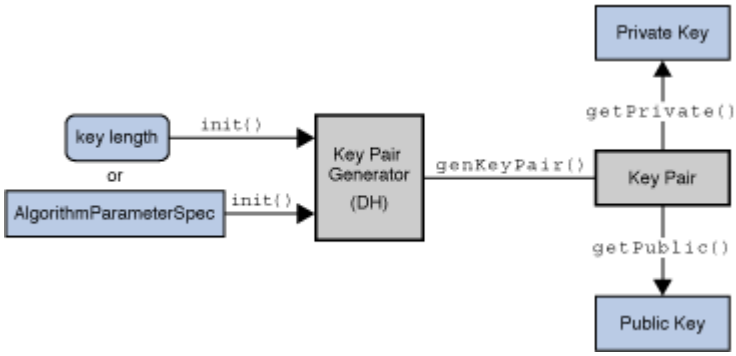
`keySpec` identifies the specification class in which the key material should be returned. It could, for example, be `DESKeySpec.class`, to indicate that the key material should be returned in an instance of the `DESKeySpec` class.

The KeyPairGenerator Class

The `KeyPairGenerator` class is an engine class (see Engine Classes and Algorithms) used to generate pairs of public and private keys.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 2-14 KeyPairGenerator Class



There are two ways to generate a key pair: in an algorithm-independent manner, and in an algorithm-specific manner. The only difference between the two is the initialization of the object.

See Generating a Pair of Keys for examples of calls to the methods of `KeyPairGenerator`.

Creating a KeyPairGenerator

All key pair generation starts with a `KeyPairGenerator`. `KeyPairGenerator` objects are obtained by using one of the `KeyPairGenerator` `getInstance()` static factory methods. See How Provider Implementations Are Requested and Supplied.

Initializing a KeyPairGenerator

A key pair generator for a particular algorithm creates a public/private key pair that can be used with this algorithm. It also associates algorithm-specific parameters with each of the generated keys.

A key pair generator needs to be initialized before it can generate keys. In most cases, algorithm-independent initialization is sufficient. But in other cases, algorithm-specific initialization can be used.

Algorithm-Independent Initialization

All key pair generators share the concepts of a keysize and a source of randomness. The keysize is interpreted differently for different algorithms. For example, in the case of the DSA algorithm, the keysize corresponds to the length of the modulus. (See Java Security Standard Algorithm Names for information about the keysizes for specific algorithms.)

An `initialize` method takes two universally shared types of arguments:

```
void initialize(int keysize, SecureRandom random)
```

Another `initialize` method takes only a `keysize` argument; it uses a system-provided source of randomness:

```
void initialize(int keysize)
```

Since no other parameters are specified when you call these algorithm-independent `initialize` methods, it is up to the provider what to do about the algorithm-specific parameters (if any) to be associated with each of the keys.

Copyright © 1993, 2026, Oracle and/or its affiliates.

If the algorithm is a "DSA" algorithm, and the modulus size (keysize) is 512, 768, 1024, 2048, or 3072, then the `SUN` provider uses a set of precomputed values for the `p`, `q`, and `g` parameters. If the modulus size is not one of these values, the `SUN` provider creates a new set of parameters. Other providers might have precomputed parameter sets for more than just the three modulus sizes mentioned previously. Still others might not have a list of precomputed parameters at all and instead always create new parameter sets.

Algorithm-Specific Initialization

For situations where a set of algorithm-specific parameters already exists (such as "community parameters" in DSA), there are two `initialize` methods that have an The AlgorithmParameterSpec Interface argument. One also has a `SecureRandom` argument, while the source of randomness is system-provided for the other:

```
void initialize(AlgorithmParameterSpec params,
                SecureRandom random)

void initialize(AlgorithmParameterSpec params)
```

See Generating a Pair of Keys.

Generating a Key Pair

The procedure for generating a key pair is always the same, regardless of initialization (and of the algorithm). You always call the following method from `KeyPairGenerator`:

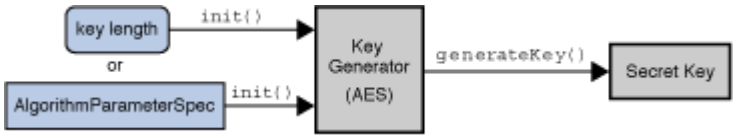
```
KeyPair generateKeyPair()
```

Multiple calls to `generateKeyPair` will yield different key pairs.

The KeyGenerator Class

A key generator is used to generate secret keys for symmetric algorithms.

Figure 2-15 The KeyGenerator Class



Creating a `KeyGenerator`

`KeyGenerator` objects are obtained by using one of the `KeyGenerator` `getInstance()` static factory methods. See How Provider Implementations Are Requested and Supplied.

Initializing a KeyGenerator Object

A key generator for a particular symmetric-key algorithm creates a symmetric key that can be used with that algorithm. It also associates algorithm-specific parameters (if any) with the generated key.

Copyright © 1993, 2026, Oracle and/or its affiliates.

There are two ways to generate a key: in an algorithm-independent manner, and in an algorithm-specific manner. The only difference between the two is the initialization of the object:

- **Algorithm-Independent Initialization**
  All key generators share the concepts of a *keysize* and a *source of randomness*. There is an `init` method that takes these two universally shared types of arguments. There is also one that takes just a `keysize` argument, and uses a system-provided source of randomness, and one that takes just a source of randomness:

  ```
  public void init(SecureRandom random);

  public void init(int keysize);

  public void init(int keysize, SecureRandom random);
  ```

  Since no other parameters are specified when you call these algorithm-independent `init` methods, it is up to the provider what to do about the algorithm-specific parameters (if any) to be associated with the generated key.
- **Algorithm-Specific Initialization**
  For situations where a set of algorithm-specific parameters already exists, there are two `init` methods that have an `AlgorithmParameterSpec` argument. One also has a `SecureRandom` argument, while the source of randomness is system-provided for the other:

  ```
  public void init(AlgorithmParameterSpec params);

  public void init(AlgorithmParameterSpec params, SecureRandom random);
  ```

In case the client does not explicitly initialize the KeyGenerator (via a call to an `init` method), each provider must supply (and document) a default initialization.

Creating a Key

The following method generates a secret key:

```
public SecretKey generateKey();
```

The KDF Class

The `KDF` class provides the functionality of a Key Derivation Function (KDF), which uses cryptographic inputs to create new cryptographically strong key material.

For more information about the Key Derivation Function API, see JEP 510.

These cryptographic inputs include input key material, an optional salt, and other parameters depending on the KDF implementation. To create one or more keys, use this material, an instance of the `KDF` class, and a cryptographic algorithm. A key is represented as a `SecretKey` object with the specified algorithm or as a byte array of raw data.

With a KDF, you can create keys that are both secure and reproducible by you and another party who share knowledge of the inputs. Deriving keys with a KDF is similar to hashing passwords. A KDF uses a keyed hash along with additional entropy from its other inputs to either extract new key material or securely expand values into a larger stream of key material.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The SunJCE provider includes KDF implementations of the HMAC-Based Extract-and-Expand Key Derivation Function (HKDF), RFC 5869.

The HKDF Algorithm

The HKDF algorithm consists of two functions: HKDF-Extract and HKDF-Expand:

- HKDF-Extract: This function takes input key material (IKM), such as shared secrets exchanged through a Diffie–Hellman key exchange, and an optional salt, which is a random value. The function produces a cryptographic key called a pseudorandom key (PRK).
- HKDF-Expand: This function takes a PRK and "info" and generates cryptographically strong key material of a specified length. "Info" is data specific to a context or an application that could include, for example, a protocol number, an algorithm identifier, or a user identity. By varying the info parameter, you can create multiple key material that use the same IKM but are bound to different applications or contexts.

Specify the parameters of the HKDF algorithm through the `HKDFParameterSpec` interface, which extends the `AlgorithmParameterSpec` interface.

Using the KDF Class

The `KDF` class has two fundamental operations:

- Instantiation and initialization, which creates the KDF and initializes it with the appropriate parameters.
- Derivation, which accepts key material and other optional inputs as well as parameters to describe the output, and then generates the derived key or data.

Instantiating a KDF and Initializing Its Parameters

To initialize a KDF, call one of the `KDF` class's `getInstance` methods and specify the key derivation algorithm to use and any other parameters. The following example initializes a KDF with the HKDF-SHA256 algorithm:

```
KDF hkdf = KDF.getInstance("HKDF-SHA256");
```

These `getInstance` methods both instantiate a `KDF` instance and initialize its algorithm. See the KDF Algorithms section in the *Java Security Standard Algorithm Names* specification for information about standard KDF algorithm names.

Deriving a Key

The `KDF` class defines two methods for deriving keys:

- `deriveData(AlgorithmParameterSpec spec)` derives a key using the specified parameters and returns it as a byte array, which can be used for entropy or if key material is required as a byte array.
- `deriveKey(String alg, AlgorithmParameterSpec spec)` derives a key using the specified parameters and returns it as a `SecretKey` object with the specified algorithm.

Use the HKDF algorithm to derive keys with these two methods. Specify parameters for this algorithm by calling methods of `HKDFParameterSpec.Builder`. Initialize a `Builder` object by calling `HKDFParameterSpec.ofExtract()`. Alternatively, specify parameters for the HKDF algorithm by calling methods of `HKDFParameterSpec.Expand`. Initialize an `Expand` object by calling `HKDFParameterSpec.expandOnly()`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

After specifying the parameters for the HKDF algorithm, call one of the following methods:

- `HKDFParameterSpec.Builder.thenExpand(byte[] info, int length)`: Creates a `HKDFParameterSpec.ExtractThenExpand` object, which contains the parameters required to perform the HKDF-Extract and HKDF-Expand functions.
- `HKDFParameterSpec.Builder.extractOnly()`: Creates a `HKDFParameterSpec.Extract` object, which contains the parameters required to perform only the HKDF-Extract function.

Both `HKDFParameterSpec.ExtractThenExpand` and `HKDFParameterSpec.Extract` are subclasses of `HKDFParameterSpec`.

Derive a key, represented as a `SecretKey` object, by calling `KDF.deriveKey` with the algorithm and the `HKDFParameterSpec` instance. Alternatively, derive key material, represented as a byte array, by calling `KDF.deriveData` with the `HKDFParameterSpec` instance.

The following example creates a `SecretKey` with the HKDF-SHA256 key derivation algorithm and the AES cryptographic algorithm:

```
public SecretKey createKeyWithKDF(byte[] salt, byte[] initialKeyMaterial,
                                   byte[] info, int keyMaterialLength)
    throws NoSuchAlgorithmException, InvalidAlgorithmParameterException {

    KDF hkdf = KDF.getInstance("HKDF-SHA256");
    AlgorithmParameterSpec params =
        HKDFParameterSpec.ofExtract()
                         .addIKM(initialKeyMaterial)
                         .addSalt(salt)
                         .thenExpand(info, keyMaterialLength);
    return hkdf.deriveKey("AES", params);
}
```

Implementing KDF Providers

To implement a KDF provider, extend the abstract class `javax.crypto.KDFSpi`.

Some KDF algorithms derive multiple cryptographic keys in a single derivation operation. If you are implementing such an algorithm, it is recommended that you either provide a `SecretKey` subclass with methods that provide access to each key, or return all of the keys in a single byte array and document how to separate them.

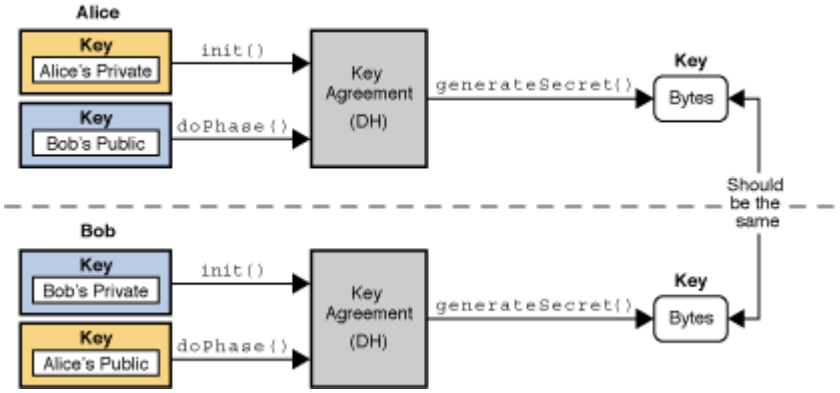
See How to Implement a Provider in the Java Cryptography Architecture.

The KeyAgreement Class

Key agreement is a protocol by which 2 or more parties can establish the same cryptographic keys, without having to exchange any secret information.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 2-16 The KeyAgreement Class



Each party initializes their key agreement object with their private key, and then enters the public keys for each party that will participate in the communication. In most cases, there are just two parties, but algorithms such as Diffie-Hellman allow for multiple parties (3 or more) to participate. When all the public keys have been entered, each `KeyAgreement` object will generate (agree upon) the same key.

The KeyAgreement class provides the functionality of a key agreement protocol. The keys involved in establishing a shared secret are created by one of the key generators (`KeyPairGenerator` or `KeyGenerator`), a `KeyFactory`, or as a result from an intermediate phase of the key agreement protocol.

Creating a KeyAgreement Object

Each party involved in the key agreement has to create a KeyAgreement object. `KeyAgreement` objects are obtained by using one of the `KeyAgreement getInstance()` static factory methods. See How Provider Implementations Are Requested and Supplied.

Initializing a KeyAgreement Object

You initialize a KeyAgreement object with your private information. In the case of Diffie-Hellman, you initialize it with your Diffie-Hellman private key. Additional initialization information may contain a source of randomness and/or a set of algorithm parameters. Note that if the requested key agreement algorithm requires the specification of algorithm parameters, and only a key, but no parameters are provided to initialize the KeyAgreement object, the key must contain the required algorithm parameters. (For example, the Diffie-Hellman algorithm uses a prime modulus `p` and a base generator `g` as its parameters.)

To initialize a KeyAgreement object, call one of its `init` methods:

```
public void init(Key key);

public void init(Key key, SecureRandom random);

public void init(Key key, AlgorithmParameterSpec params);

public void init(Key key, AlgorithmParameterSpec params,
                 SecureRandom random);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Executing a KeyAgreement Phase

Every key agreement protocol consists of a number of phases that need to be executed by each party involved in the key agreement.

To execute the next phase in the key agreement, call the `doPhase` method:

```
public Key doPhase(Key key, boolean lastPhase);
```

The `key` parameter contains the key to be processed by that phase. In most cases, this is the public key of one of the other parties involved in the key agreement, or an intermediate key that was generated by a previous phase. `doPhase` may return an intermediate key that you may have to send to the other parties of this key agreement, so they can process it in a subsequent phase.

The `lastPhase` parameter specifies whether or not the phase to be executed is the last one in the key agreement: A value of `FALSE` indicates that this is not the last phase of the key agreement (there are more phases to follow), and a value of `TRUE` indicates that this is the last phase of the key agreement and the key agreement is completed, i.e., `generateSecret` can be called next.

In the example of Diffie-Hellman Key Exchange between Two Parties , you call `doPhase` once, with `lastPhase` set to `TRUE`. In the example of Diffie-Hellman between three parties, you call `doPhase` twice: the first time with `lastPhase` set to `FALSE`, the 2nd time with `lastPhase` set to `TRUE`.

Generating the Shared Secret

After each party has executed all the required key agreement phases, it can compute the shared secret by calling one of the `generateSecret` methods:

```
public byte[] generateSecret();

public int generateSecret(byte[] sharedSecret, int offset);

public SecretKey generateSecret(String algorithm);
```

Key Management

A database called a "keystore" can be used to manage a repository of keys and certificates. (A *certificate* is a digitally signed statement from one entity, saying that the public key of some other entity has a particular value.)

Keystore Location

The user keystore is by default stored in a file named `.keystore` in the user's home directory, as determined by the `user.home` system property whose default value depends on the operating system:

- Linux and macOS: `/home/`*`username`*`/`
- Windows: `C:\Users\`*`username`*`\`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Of course, keystore files can be located as desired. In some environments, it may make sense for multiple keystores to exist. For example, one keystore might hold a user's private keys, and another might hold certificates used to establish trust relationships.

In addition to the user's keystore, the JDK also maintains a system-wide keystore which is used to store trusted certificates from a variety of Certificate Authorities (CA's). These CA certificates can be used to help make trust decisions. For example, in SSL/TLS/DTLS when the `SunJSSE` provider is presented with certificates from a remote peer, the default trustmanager will consult one of the following files to determine if the connection is to be trusted:

- Linux and macOS: `<java-home>/lib/security/cacerts`
- Windows: `<java-home>\lib\security\cacerts`

Instead of using the system-wide `cacerts` keystore, applications can set up and use their own keystores, or even use the user keystore described previously.

Keystore Implementation

The `KeyStore` class supplies well-defined interfaces to access and modify the information in a keystore. It is possible for there to be multiple different concrete implementations, where each implementation is that for a particular *type* of keystore.

Currently, there are two command-line tools that make use of `KeyStore`: **`keytool`** and **`jarsigner`**. Because `KeyStore` is publicly available, JDK users can write additional security applications that use it.

Applications can choose different *types* of keystore implementations from different providers, using the `getInstance` factory method in the `KeyStore` class. A keystore type defines the storage and data format of the keystore information, and the algorithms used to protect private keys in the keystore and the integrity of the keystore itself. Keystore implementations of different types are not compatible.

The default keystore implementation is "pkcs12". This is a cross-platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This standard is primarily meant for storing or transporting a user's private keys, certificates, and miscellaneous secrets. Arbitrary attributes can be associated with individual entries in a PKCS12 keystore.

```
keystore.type=pkcs12
```

To have tools and other applications use a different default keystore implementation, you can change that line to specify a different type.

Some applications, such as `keytool`, also let you override the default keystore type (via the `-storetype` command-line parameter).

> **Note**
>
> Keystore type designations are case-insensitive. For example, "pkcs12" would be considered the same as "PKCS12".

PKCS12 is the default and recommended keystore type. However, there are three other types of keystores that come with the JDK implementation.

1. **"jceks"** is an alternate proprietary keystore format to "jks" that uses Password-Based Encryption with Triple-DES.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The "jceks" implementation can parse and convert a "jks" keystore file to the "jceks" format. You may upgrade your keystore of type "jks" to a keystore of type "jceks" by changing the password of a private-key entry in your keystore and specifying "`-storetype jceks`" as the keystore type. To apply the cryptographically strong(er) key protection supplied to a private key named "signkey" in your default keystore, use the following command, which will prompt you for the old and new key passwords:

```
keytool -keypass -alias signkey -storetype jceks
```

See `keytool` in *Java Development Kit Tool Specifications* .

2. **"jks"** is another option. It implements the keystore as a file, utilizing a proprietary keystore type (format). It protects each private key with its own individual password, and also protects the integrity of the entire keystore with a (possibly different) password.
3. **"dks"** is a domain keystore. It is a collection of keystores presented as a single logical keystore. The keystores that comprise a given domain are specified by configuration data whose syntax is described in `DomainLoadStoreParameter`.

> **Note**
>
> JKS and JCEKS keystores will be removed in a future release because they use outdated cryptographic algorithms. It is advised that you migrate to a PKCS12 keystore by using the `keytool -importkeystore` command.

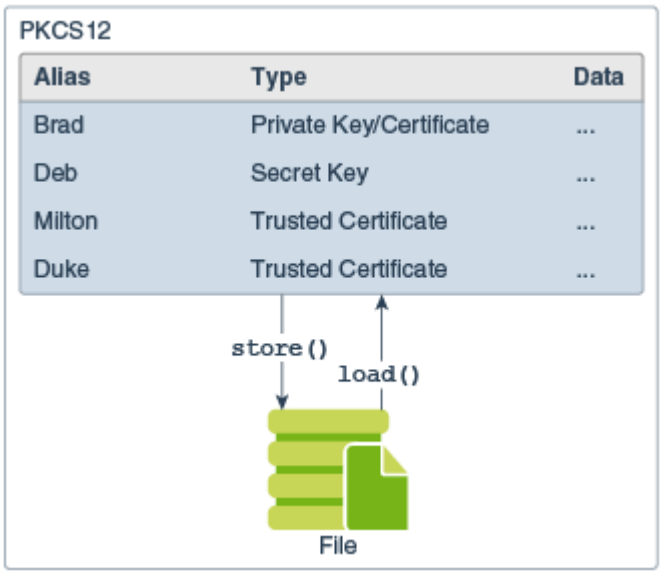
Keystore implementations are provider-based. If you want to write your own KeyStore implementations, see How to Implement a Provider in the Java Cryptography Architecture.

The KeyStore Class

The `KeyStore` class supplies well-defined interfaces to access and modify the information in a keystore.

The `KeyStore` class is an Engine Classes and Algorithms.

Figure 2-17 KeyStore Class



Copyright © 1993, 2026, Oracle and/or its affiliates.

This class represents an in-memory collection of keys and certificates. `KeyStore` manages two types of entries:

- **Key Entry**: This type of keystore entry holds very sensitive cryptographic key information, which must be protected from unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain authenticating the corresponding public key.

  Private keys and certificate chains are used by a given entity for self-authentication using digital signatures. For example, software distribution organizations digitally sign JAR files as part of releasing and/or licensing software.

- **Trusted Certificate Entry**: This type of entry contains a single public key certificate belonging to another party. It is called a *trusted certificate* because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the *subject* (owner) of the certificate.

  This type of entry can be used to authenticate other parties.

Each entry in a keystore is identified by an "alias" string. In the case of private keys and their associated certificate chains, these strings distinguish among the different ways in which the entity may authenticate itself. For example, the entity may authenticate itself using different certificate authorities, or using different public key algorithms.

Whether keystores are persistent, and the mechanisms used by the keystore if it is persistent, are not specified here. This convention allows use of a variety of techniques for protecting sensitive (e.g., private or secret) keys. Smart cards or other integrated cryptographic engines (SafeKeyper) are one option, and simpler mechanisms such as files may also be used (in a variety of formats).

The following describes the main `KeyStore` methods.

Creating a KeyStore Object

KeyStore objects are obtained by using one of the KeyStore getInstance() method. See How Provider Implementations Are Requested and Supplied.

Loading a Particular Keystore into Memory

Before a `KeyStore` object can be used, the actual keystore data must be loaded into memory via the `load` method:

```
final void load(InputStream stream, char[] password)
```

The optional password is used to check the integrity of the keystore data. If no password is supplied, no integrity check is performed.

To create an empty keystore, you pass `null` as the `InputStream` argument to the `load` method.

A DKS keystore is loaded by passing a `DomainLoadStoreParameter` to the alternative load method:

```
final void load(KeyStore.LoadStoreParameter param)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Getting a List of the Keystore Aliases

All keystore entries are accessed via unique *aliases*. The `aliases` method returns an enumeration of the alias names in the keystore:

```
final Enumeration aliases()
```

Determining Keystore Entry Types

As stated in the KeyStore class, there are two different types of entries in a keystore. The following methods determine whether the entry specified by the given alias is a key/certificate or a trusted certificate entry, respectively:

```
final boolean isKeyEntry(String alias)
final boolean isCertificateEntry(String alias)
```

Adding/Setting/Deleting Keystore Entries

The `setCertificateEntry` method assigns a certificate to a specified alias:

```
final void setCertificateEntry(String alias, Certificate cert)
```

If `alias` doesn't exist, a trusted certificate entry with that alias is created. If `alias` exists and identifies a trusted certificate entry, the certificate associated with it is replaced by `cert`.

The `setKeyEntry` methods add (if `alias` doesn't yet exist) or set key entries:

```
final void setKeyEntry(String alias,
                       Key key,
                       char[] password,
                       Certificate[] chain)

final void setKeyEntry(String alias,
                       byte[] key,
                       Certificate[] chain)
```

In the method with `key` as a byte array, it is the bytes for a key in protected format. For example, in the keystore implementation supplied by the `SUN` provider, the `key` byte array is expected to contain a protected private key, encoded as an `EncryptedPrivateKeyInfo` as defined in the PKCS8 standard. In the other method, the `password` is the password used to protect the key.

The `deleteEntry` method deletes an entry:

```
final void deleteEntry(String alias)
```

PKCS #12 keystores support entries containing arbitrary attributes. Use the `PKCS12Attribute` class to create the attributes. When creating the new keystore entry use a constructor method that accepts attributes. Finally, use the following method to add the entry to the keystore:

```
final void setEntry(String alias, Entry entry,
                    ProtectionParameter protParam)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Getting Information from the Keystore

The `getKey` method returns the key associated with the given alias. The key is recovered using the given password:

```
final Key getKey(String alias, char[] password)
```

The following methods return the certificate, or certificate chain, respectively, associated with the given alias:

```
final Certificate getCertificate(String alias)
final Certificate[] getCertificateChain(String alias)
```

You can determine the name (`alias`) of the first entry whose certificate matches a given certificate via the following:

```
final String getCertificateAlias(Certificate cert)
```

PKCS #12 keystores support entries containing arbitrary attributes. Use the following method to retrieve an entry that may contain attributes:

```
final Entry getEntry(String alias, ProtectionParameter protParam)
```

and then use the `KeyStore.Entry.getAttributes` method to extract such attributes and use the methods of the `KeyStore.Entry.Attribute` interface to examine them.

Saving the KeyStore

The in-memory keystore can be saved via the `store` method:

```
final void store(OutputStream stream, char[] password)
```

The password is used to calculate an integrity checksum of the keystore data, which is appended to the keystore data.

A DKS keystore is stored by passing a `DomainLoadStoreParameter` to the alternative store method:

```
final void store(KeyStore.LoadStoreParameter param)
```

The BinaryEncodable Interface

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Language and VM Features.
> For more information about the `BinaryEncodable` interface and the Java SE API classes that implement it, see JEP 538.

Cryptographic API classes that implement the `BinaryEncodable` interface represent cryptographic objects with binary-encodable or certificate material.

A binary-encodable cryptographic object can be encoded to and decoded from the widely-used The Privacy-Enhanced Mail (PEM) Transport Format.

The `BinaryEncodable` interface is sealed and permits the following classes and interfaces:

- `AsymmetricKey`: This interface contains methods that are common to either a public key or a private key.
- `EncryptedPrivateKeyInfo`: This class implements the `EncryptedPrivateKeyInfo` type as defined in Public-Key Cryptography Standards (PKCS) #8. You can use an instance of this class to decrypt and produce a `PrivateKey` object.
- `KeyPair`: This class is a simple holder for a key pair (a public key and a private key). See The KeyPair Class.
- `PEM`: This class implements the `BinaryEncodable` interface and represents PEM data by its type, Base64-encoded content, and optional leading data. The `PEMEncoder` and `PEMDecoder` classes use `PEM` when representing the data as a cryptographic object is not desired or the type has no `BinaryEncodable`.
- `PKCS8EncodedKeySpec`: This class represents the Distinguished Encoding Rules (DER) encoding of a private key according to the format specified in the PKCS #8 standard. DER is a serialization format (a way to convert a data structure to bytes and back again) for the Abstract Syntax Notation One (ASN.1) format, which is a standard, algorithm-independent encoding format used extensively in cryptography. See The PKCS8EncodedKeySpec Class.
- `X509EncodedKeySpec`: This class represents the DER encoding of a public key, according to the format specified in the X.509 standard. See The X509EncodedKeySpec Class.
- `X509Certificate`: This abstract class provides a standard way to access the attributes of X.509 certificates. See X509Certificate Class.
- `X509CRL`: This is an abstract class for an X.509 Certificate Revocation List (CRL). See The CertificateFactory Class.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The list of permitted subtypes in the `BinaryEncodable` interface's `permits` clause is not exhaustive. This means that `BinaryEncodable` has at least one permitted subtype that is not `public` and therefore is not shown in the list. If the type of the selector expression of a `switch` statement or expression is `BinaryEncodable`, you cannot make the `switch` exhaustive by providing a `case` label for every permitted subtype shown in the list. You must include a `default` or `case BinaryEncodable` label to handle additional subtypes. This allows the list of permitted subtypes to change over time without causing preexisting `switch` statements or expressions to fail because of an unrecognized subtype.

Although many cryptography standards use binary encodings to serialize cryptographic objects, some systems only support ASCII when sending and receiving data. These systems typically use the PEM transport format. An instance of a class that implements `BinaryEncodable` can be encoded to, and decoded from, the PEM format with the `PEMEncoder` and `PEMDecoder` classes.

The Privacy-Enhanced Mail (PEM) Transport Format

The Privacy-Enhanced Mail (PEM) transport format is a textual format for binary data. Applications often use this format to send and receive representations of cryptographic objects because some systems, such as e-mail, support only ASCII.

The PEM format was originally designed for sending cryptographic objects through e-mail. Over time, it has been used and extended for other purposes such as the following:

- Certificate authorities issue certificate chains in the PEM format.
- Cryptographic libraries such as OpenSSL provide operations for generating and converting PEM-encoded cryptographic objects.
- Security-sensitive applications such as OpenSSH store communication keys in the PEM format.
- Hardware authentication devices such as Yubikeys ingest and dispense PEM-encoded cryptographic objects.

To encode to and decode from the PEM format, use the `PEMEncoder` and `PEMDecoder` classes.

The PEMEncoder Class

Use an instance of the `PEMEncoder` class to encode a `BinaryEncodable` object, such as a `PrivateKey`, `KeyPair`, or a `PKCS8EncodedKeySpec` object, to the PEM format.

The following example encodes a `PrivateKey` to a PEM-formatted byte array:

```
byte[] encodePrivateKeyToPEM(PrivateKey privateKey) {
    PEMEncoder pe = PEMEncoder.of();
    return pe.encode(privateKey);
}
```

You can reuse the `PEMEncoder` instance returned by the `PEMEncoder.of()` factory method.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The following example encodes a `PrivateKey` to a PEM-formatted string:

```
String encodePrivateKeyToString(PEMEncoder pe, PrivateKey privateKey) {
    return pe.encodeToString(privateKey);
}
```

The following example encodes a public/private key pair into a string:

```
String encodeKeyPairToString(PEMEncoder pe, PublicKey pub, PrivateKey priv) {
    return pe.encodeToString(new KeyPair(pub, priv));
}
```

If you are encoding a `PrivateKey`, then you can encrypt it (see “Textual Encoding of PKCS #8 Encrypted Private Key Info” in RFC 7468, “Textual Encodings of PKIX, PKCS, and CMS Structures) with the `PEMEncoder.withEncryption(char[])` method, which takes a password and returns a new immutable `PEMEncoder` instance configured to encrypt the key with that password. The following example consists of two methods. The first method, `encryptPEMEncoder`, returns a `PEMEncoder` configured to encrypt a key with the given password. The second method, `decodePrivateKeyWithPassword`, decrypts the `PrivateKey` that has been encoded by `encryptedPEMEncoder`:

```
PEMEncoder encryptedPEMEncoder(String password) {
    return PEMEncoder.of().withEncryption(password.toCharArray());
}

PrivateKey decodePrivateKeyWithPassword(
    String encryptedPrivateKey, String password) {
    return (PrivateKey) PEMDecoder.of()
        .withDecryption(password.toCharArray())
        .decode(encryptedPrivateKey);
}
```

The method `PEMDecoder.decode(String)` throws an `IllegalArgumentException` if there's a decoding error or the PEM data is unsupported. See The PEMDecoder Class for more information about decoding PEM-formatted objects.

The PEMDecoder Class

Use an instance of the `PEMDecoder` class to decode a `BinaryEncodable` object from PEM-formatted data:

```
BinaryEncodable be = PEMDecoder.of().decode(pem);
```

You can reuse the `PEMDecoder` instance returned by the `PEMDecoder.of()` factory method.

You can determine the type of a `BinaryEncodable` with pattern matching with `instanceof` or `switch`; see Pattern Matching in *Java Platform, Standard Edition Java Language Updates*:

```
PEMDecoder pd = PEMDecoder.of();
switch (pd.decode(pem)) {
    case PublicKey publicKey -> ...;
    case PrivateKey privateKey -> ...;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    case KeyPair keyPair -> ...;
    case PKCS8EncodedKeySpec pkcs8Key -> ...;
    default -> throw new IllegalArgumentException(...);
}
```

If you know the type of the `BinaryEncodable`, you can specify it as an argument for `PEMDecoder.decode(InputStream, Class)` or `PEMDecoder.decode(String, Class)`, for example:

```
ECPrivateKey decodedECPrivateKey = pd.decode(privateKeyPEM,
ECPrivateKey.class);
```

As mentioned in The PEMEncoder Class, if you are encoding a `PrivateKey`, then you can encrypt it with the `PEMEncoder.withEncryption(char[])` method, which takes a password and returns a new immutable `PEMEncoder` instance configured to encrypt the key with that password. You can then decrypt it with the `PEMDecoder.withDecryption(char[])` method, which takes a password and returns a new immutable `PEMDecoder` configured to encrypt the key with that password. For example:

```
PrivateKey decodePrivateKeyWithPassword(
    String encryptedPrivateKey, String password) {
    return (PrivateKey) PEMDecoder.of()
        .withDecryption(password.toCharArray())
        .decode(encryptedPrivateKey);
}
```

Instead of casting the result of `decode` to the expected data type, you can specify it as an argument:

```
ECPrivateKey decodeECPrivateKeyWithPassword(
    String encryptedPrivateKey, String password) {
    return PEMDecoder.of()
        .withDecryption(password.toCharArray())
        .decode(encryptedPrivateKey, ECPrivateKey.class);
}
```

Sometimes, you may need a specific cryptographic provider to decode PEM-formatted data. You can call the `PEMDecoder.withFactoriesOf(Provider)` method, which returns a new `PEMDecoder` instance that uses the specified provider to produce cryptographic objects. The following example decodes a `Certificate` with a hypothetical provider named `org.example.MyProvider` (which is created by calling its constructor, `MyProvider()`):

```
// Dynamically register the org.example.MyProvider provider
Security.addProvider(new MyProvider());
// Create a PEMDecoder that uses MyProvider
PEMDecoder d = pd.withFactoriesOf(Security.getProvider("MyProvider"));
// Decode the Certificate
Certificate c = d.decode(pem, X509Certificate.class);
```

The EncryptedPrivateKeyInfo Class

The `EncryptedPrivateKeyInfo` class represents an encrypted private key.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The static method `EncryptedPrivateKeyInfo.encrypt(BinaryEncodable, char[])` encrypts the given `BinaryEncodable` with the given password. The `BinaryEncodable` must be a `PrivateKey`, `KeyPair`, or a `PKCS8EncodedKeySpec`.

With the static method `EncryptedPrivateKeyInfo.encrypt(BinaryEncodable, char[], String, AlgorithmParameterSpec, Provider)` you can specify which password-based encryption (PBE) algorithm and associated parameters to use to encrypt the `BinaryEncodable`. See `Cipher` Algorithms in *Java Security Standard Algorithm Names* for information about the PBE algorithm string format. See The SunJCE Provider for a list of PBE algorithms supported by the SunJCE provider. See Algorithm Parameter Specification Interfaces and Classes for information about specifying cryptographic parameters, including those for the `PBEParameterSpec` class. See Using Password-Based Encryption for an example that uses the `PBEParameterSpec` class.

The following example encrypts a private key with `EncryptedPrivateKeyInfo.encrypt`, then uses the returned `EncryptedPrivateKeyInfo` instance to encode it to PEM-formatted text:

```
String password = ...;
var epki = EncryptedPrivateKeyInfo.encrypt(privateKey,
password.toCharArray());
String pemEncyptedPrivateKey = PEMEncoder.of().encodeToString(epki);
```

The `getKey` methods decrypt the private key in an `EncryptedPrivateKeyInfo` instance. The following example decrypts a PEM-formatted `PrivateKey` and then uses the returned `EncryptedPrivateKeyInfo` instance to decode it to a `PrivateKey`:

```
EncryptedPrivateKeyInfo epkiDecoded =
    PEMDecoder.of().decode(pemEncyptedPrivateKey,
EncryptedPrivateKeyInfo.class);
PrivateKey key = epkiDecoded.getKey(password.toCharArray());
```

The `getKeyPair` methods decrypt an `EncryptedPrivateKeyInfo` instance that contains both a public key and a private key. In the following example, `getKeyPairWithPassword` demonstrates `getKeyPair(char[])` while `getKeyPairWithKey` demonstrates `getKeyPair(Key)`:

```
// Demonstrates EncryptedPrivateKeyInfo.getKeyPair(char[])
KeyPair getKeyPairWithPassword(KeyPair original, char[] password)
        throws Exception {
    EncryptedPrivateKeyInfo encrypted =
        EncryptedPrivateKeyInfo.encrypt(original, password);

    String pem = PEMEncoder.of().encodeToString(encrypted);

    EncryptedPrivateKeyInfo decoded = PEMDecoder.of()
        .decode(pem, EncryptedPrivateKeyInfo.class);
    return decoded.getKeyPair(password);
}

// Demonstrates EncryptedPrivateKeyInfo.getKeyPair(Key)
KeyPair getKeyPairWithKey(KeyPair original, char[] password)
            throws Exception {
    EncryptedPrivateKeyInfo encrypted =
        EncryptedPrivateKeyInfo.encrypt(original, password);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        String pem = PEMEncoder.of().encodeToString(encrypted);

        EncryptedPrivateKeyInfo decoded = PEMDecoder.of()
            .decode(pem, EncryptedPrivateKeyInfo.class);

        SecretKey key = pbeKey(decoded.getAlgName(), password);
        return decoded.getKeyPair(key);
    }

    // Derives a password-based encryption (PBE) key
    SecretKey pbeKey(String algorithm, char[] password) throws Exception {
        PBEKeySpec keySpec = new PBEKeySpec(password);
        try {
            return
    SecretKeyFactory.getInstance(algorithm).generateSecret(keySpec);
        } finally {
            keySpec.clearPassword();
        }
    }

    // Creates an EC key pair
    KeyPair generateSigningKeyPair() throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC");
        kpg.initialize(new ECGenParameterSpec("secp256r1"));
        return kpg.generateKeyPair();
    }
```

The method `getKeyPairWithPassword` encrypts a key pair with a password and the default password-based encryption (PBE) algorithm, encodes it as a PEM, decodes it back to an `EncryptedPrivateKeyInfo`, and recovers the key pair with `getKeyPair(char[])`. The method `getKeyPairWithKey` creates a key pair with a password-based encryption (PBE) key, encodes it as a PEM, decodes it back to an `EncryptedPrivateKeyInfo`, derives a key based on the encryption algorithm of the `EncryptedPrivateKeyInfo`, and recovers the key pair with `getKeyPair(Key)`.

> **Note**
>
> The default password-based encryption (PBE) algorithm used when encrypting a `BinaryEncodable` with either `PEMEncoder` or `EncryptedPrivateKeyInfo` is defined in the default master security properties file (see The Security Properties File) in the `jdk.epkcs8.defaultAlgorithm` security property. The default algorithm might change in the future, but this will not affect PEM text created now because the data encoded in that text contains the algorithm name and all other parameters necessary for decryption.

The PEM Class

Use the `PEM` class, which implements the `BinaryEncodable` interface, to encode and decode PEM texts that represent cryptographic objects for which no Java Platform API exist, for example, PKCS #10 certification requests.

Copyright © 1993, 2026, Oracle and/or its affiliates.

An instance of `PEMDecoder` decodes PEM text into a `PEM` object when there is no Java Platform API for the PEM text's type. The following example decodes PEM text stored in a file, but throws an `IllegalArgumentException` if the PEM text's type is `PEM`:

```
BinaryEncodable decodeFile(File f) throws IOException,
IllegalArgumentException {
    BinaryEncodable d = null;
    try (InputStream is = new FileInputStream(f)) {
        d = PEMDecoder.of().decode(is);
        if (d instanceof PEM pr) {
            throw new IllegalArgumentException(
                "Unhandled PEM type: " + pr.type()
                + "\nData: " + pr.pem());
        }
    }
    return d;
}
```

If you want to access the leading data of a PEM text or handle the PEM text's content yourself, then you can specifically request a `PEM` when decoding:

```
PEM pr = PEMDecoder.of().decode(pem, PEM.class);
```

A `PEMEncoder` instance encodes a `PEM` object into PEM text without validating its content.

Algorithm Parameters Classes

Like `Key`s and `KeySpec`s, an algorithm's initialization parameters are represented by either `AlgorithmParameters` or `AlgorithmParameterSpec`s.

Depending on the use situation, algorithms can use the parameters directly, or the parameters might need to be converted into a more portable format for transmission or storage.

A *transparent* representation of a set of parameters (through `AlgorithmParameterSpec`) means that you can access each parameter value in the set individually. You can access these values through one of the `get` methods defined in the corresponding specification class (for example, `DSAParameterSpec` defines `getP`, `getQ`, and `getG` methods, to access `p`, `q`, and `g`, respectively).

In contrast, the `AlgorithmParameters` class supplies an *opaque* representation, in which you have no direct access to the parameter fields. You can only get the name of the algorithm associated with the parameter set (through `getAlgorithm`) and some kind of encoding for the parameter set (through `getEncoded`).

The AlgorithmParameterSpec Interface

`AlgorithmParameterSpec` is an interface to a transparent specification of cryptographic parameters. This interface contains no methods or constants. Its only purpose is to group (and provide type safety for) all parameter specifications. All parameter specifications must implement this interface.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The AlgorithmParameters Class

The `AlgorithmParameters` class is an engine class that provides an opaque representation of cryptographic parameters. You can initialize the `AlgorithmParameters` class using a specific `AlgorithmParameterSpec` object, or by encoding the parameters in a recognized format. You can retrieve the resulting specification with the `getParameterSpec` method.

Creating an AlgorithmParameters Object

`AlgorithmParameters` objects are obtained by using one of the `AlgorithmParameters` `getInstance()` static factory methods. For more information, see How Provider Implementations Are Requested and Supplied.

Initializing an AlgorithmParameters Object

Once an `AlgorithmParameters` object is instantiated, it must be initialized via a call to `init`, using an appropriate parameter specification or parameter encoding:

```
void init(AlgorithmParameterSpec paramSpec)
void init(byte[] params)
void init(byte[] params, String format)
```

In these `init` methods, `params` is an array containing the encoded parameters, and `format` is the name of the decoding format. In the `init` method with a `params` argument but no `format` argument, the primary decoding format for parameters is used. The primary decoding format is ASN.1, if an ASN.1 specification for the parameters exists.

Obtaining the Encoded Parameters

A byte encoding of the parameters represented in an `AlgorithmParameters` object may be obtained via a call to `getEncoded`:

```
byte[] getEncoded()
```

This method returns the parameters in their primary encoding format. The primary encoding format for parameters is ASN.1, if an ASN.1 specification for this type of parameters exists.

If you want the parameters returned in a specified encoding format, use

```
byte[] getEncoded(String format)
```

If `format` is null, the primary encoding format for parameters is used, as in the other `getEncoded` method.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Converting an AlgorithmParameters Object to a Transparent Specification

A transparent parameter specification for the algorithm parameters may be obtained from an `AlgorithmParameters` object via a call to `getParameterSpec`:

```
AlgorithmParameterSpec getParameterSpec(Class paramSpec)
```

`paramSpec` identifies the specification class in which the parameters should be returned. The specification class could be, for example, `DSAParameterSpec.class` to indicate that the parameters should be returned in an instance of the `DSAParameterSpec` class. (This class is in the `java.security.spec` package.)

The AlgorithmParameterGenerator Class

The `AlgorithmParameterGenerator` class is an Engine Classes and Algorithms used to generate a set of **brand-new** parameters suitable for a certain algorithm (the algorithm is specified when an `AlgorithmParameterGenerator` instance is created). This object is used when you do not have an existing set of algorithm parameters, and want to generate one from scratch.

Creating an AlgorithmParameterGenerator Object

`AlgorithmParameterGenerator` objects are obtained by using one of the `AlgorithmParameterGenerator` getInstance() static factory methods. See How Provider Implementations Are Requested and Supplied.

Initializing an AlgorithmParameterGenerator Object

The `AlgorithmParameterGenerator` object can be initialized in two different ways: an algorithm-independent manner or an algorithm-specific manner.

The algorithm-independent approach uses the fact that all parameter generators share the concept of a "size" and a source of randomness. The measure of size is universally shared by all algorithm parameters, though it is interpreted differently for different algorithms. For example, in the case of parameters for the DSA algorithm, "size" corresponds to the size of the prime modulus, in bits. (See Java Security Standard Algorithm Names to know more about the sizes for specific algorithms.) When using this approach, algorithm-specific parameter generation values--if any--default to some standard values. One `init` method that takes these two universally shared types of arguments:

```
void init(int size, SecureRandom random);
```

Another `init` method takes only a `size` argument and uses a system-provided source of randomness:

```
void init(int size)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

A third approach initializes a parameter generator object using algorithm-specific semantics, which are represented by a set of algorithm-specific parameter generation values supplied in an `AlgorithmParameterSpec` object:

```
void init(AlgorithmParameterSpec genParamSpec,
                          SecureRandom random)

void init(AlgorithmParameterSpec genParamSpec)
```

To generate Diffie-Hellman system parameters, for example, the parameter generation values usually consist of the size of the prime modulus and the size of the random exponent, both specified in number of bits.

Generating Algorithm Parameters

Once you have created and initialized an AlgorithmParameterGenerator object, you can use the generateParameters method to generate the algorithm parameters:

```
AlgorithmParameters generateParameters()
```

The CertificateFactory Class

The `CertificateFactory` class defines the functionality of a certificate factory, which is used to generate certificate and certificate revocation list (CRL) objects from their encoding.

The `CertificateFactory` class is an Engine Classes and Algorithms.

A certificate factory for X.509 must return certificates that are an instance of `java.security.cert.X509Certificate`, and CRLs that are an instance of `java.security.cert.X509CRL`.

Creating a CertificateFactory Object

`CertificateFactory` objects are obtained by using one of the `getInstance()` static factory methods. For more information, see How Provider Implementations Are Requested and Supplied.

Generating Certificate Objects

To generate a certificate object and initialize it with the data read from an input stream, use the `generateCertificate` method:

```
final Certificate generateCertificate(InputStream inStream)
```

To return a (possibly empty) collection view of the certificates read from a given input stream, use the `generateCertificates` method:

```
final Collection generateCertificates(InputStream inStream)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Generating CRL Objects

To generate a certificate revocation list (CRL) object and initialize it with the data read from an input stream, use the `generateCRL` method:

```
final CRL generateCRL(InputStream inStream)
```

To return a (possibly empty) collection view of the CRLs read from a given input stream, use the `generateCRLs` method:

```
final Collection generateCRLs(InputStream inStream)
```

Generating CertPath Objects

The certificate path builder and validator for PKIX is defined by the Internet X.509 Public Key Infrastructure Certificate and CRL Profile, RFC 5280.

A certificate store implementation for retrieving certificates and CRLs from Collection and LDAP directories, using the PKIX LDAP V2 Schema is also available from the IETF as RFC 2587.

To generate a `CertPath` object and initialize it with data read from an input stream, use one of the following `generateCertPath` methods (with or without specifying the encoding to be used for the data):

```
final CertPath generateCertPath(InputStream inStream)

final CertPath generateCertPath(InputStream inStream,
                                String encoding)
```

To generate a `CertPath` object and initialize it with a list of certificates, use the following method:

```
final CertPath generateCertPath(List certificates)
```

To retrieve a list of the `CertPath` encoding supported by this certificate factory, you can call the `getCertPathEncodings` method:

```
final Iterator getCertPathEncodings()
```

The default encoding will be listed first.

Standard Names

The Standard Names document contains information about the algorithm specifications.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Java Security Standard Algorithm Names describes the standard names for algorithms, certificate and keystore types that the JDK Security API requires and uses. It also contains more information about the algorithm specifications. Specific provider information can be found in JDK Providers Documentation.

Cryptographic implementations in the JDK are distributed through several different providers primarily for historical reasons (`Sun`, `SunJSSE`, `SunJCE`, `SunRsaSign`). Note these providers may not be available on all JDK implementations, and therefore, truly portable applications should call `getInstance()` without specifying specific providers. Applications specifying a particular provider may not be able to take advantage of native providers tuned for an underlying operating environment (such as PKCS or Microsoft's CAPI).

The `SunPKCS11` provider itself does not contain any cryptographic algorithms, but instead, directs requests into an underlying PKCS11 implementation. Consult the PKCS#11 Reference Guide and the underlying PKCS11 implementation to determine if a desired algorithm will be available through the PKCS11 provider. Likewise, on Windows systems, the `SunMSCAPI` provider does not provide any cryptographic functionality, but instead routes requests to the underlying operating system for handling.

How the JCA Might Be Used in a SSL/TLS Implementation

With an understanding of the JCA classes, consider how these classes might be combined to implement an advanced network protocol like SSL/TLS.

The SSL/TLS Overview section in the TLS and DTLS Protocols describes at a high level how the protocols work. As asymmetric (public key) cipher operations are much slower than symmetric operations (secret key), public key cryptography is used to establish secret keys which are then used to protect the actual application data. Vastly simplified, the SSL/TLS handshake involves exchanging initialization data, performing some public key operations to arrive at a secret key, and then using that key to encrypt further traffic.

> **Note**
>
> The details presented here simply show how some of these classes might be employed. This section will not present sufficient information for building a SSL/TLS implementation. For more information, see Java Secure Socket Extension (JSSE) Reference Guide and RFC 5246: The Transport Layer Security (TLS) Protocol, Version 1.2.

Assume that this SSL/TLS implementation will be made available as a JSSE provider. A concrete implementation of the `Provider` class is first written that will eventually be registered in the `Security` class' list of providers. This provider mainly provides a mapping from algorithm names to actual implementation classes. (that is: "SSLContext.TLS"->"com.foo.TLSImpl") When an application requests an "TLS" instance (via `SSLContext.getInstance("TLS")`), the provider's list is consulted for the requested algorithm, and an appropriate instance is created.

Before discussing details of the actual handshake, a quick review of some of the JSSE's architecture is needed. The heart of the JSSE architecture is the `SSLContext`. The context eventually creates end objects (`SSLSocket` and `SSLEngine`) which actually implement the SSL/TLS protocol. `SSLContexts` are initialized with two callback classes, `KeyManager` and `TrustManager`, which allow applications to first select authentication material to send and second to verify credentials sent by a peer.

A JSSE `KeyManager` is responsible for choosing which credentials to present to a peer. Many algorithms are possible, but a common strategy is to maintain a RSA or DSA public/private key

Copyright © 1993, 2026, Oracle and/or its affiliates.

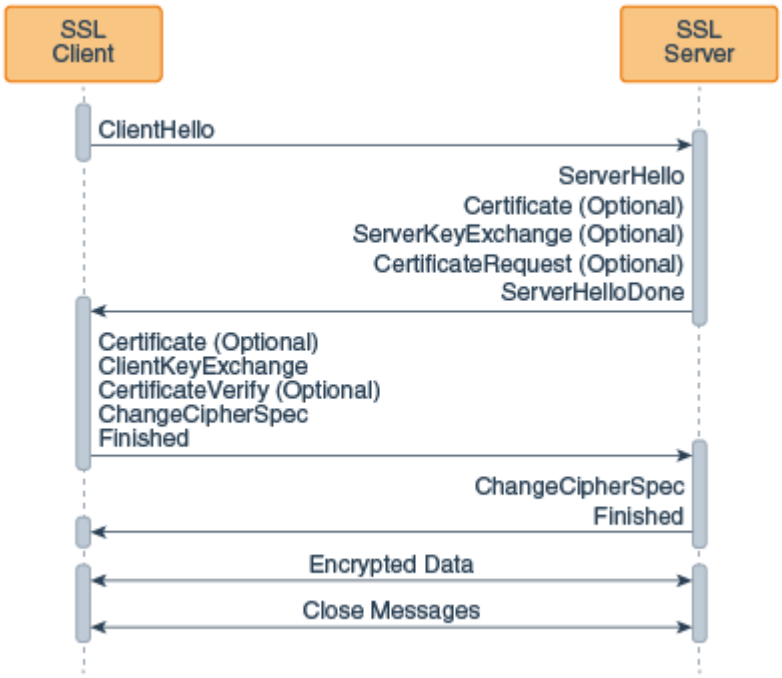
pair along with a `X509Certificate` in a `KeyStore` backed by a disk file. When a `KeyStore` object is initialized and loaded from the file, the file's raw bytes are converted into `PublicKey` and `PrivateKey` objects using a `KeyFactory`, and a certificate chain's bytes are converted using a `CertificateFactory`. When a credential is needed, the `KeyManager` simply consults this `KeyStore` object and determines which credentials to present.

A `KeyStore`'s contents might have originally been created using a utility such as `keytool`. `keytool` creates a RSA or DSA `KeyPairGenerator` and initializes it with an appropriate keysize. This generator is then used to create a `KeyPair` which `keytool` would store along with the newly-created certificate in the `KeyStore`, which is eventually written to disk.

A JSSE `TrustManager` is responsible for verifying the credentials received from a peer. There are many ways to verify credentials: one of them is to create a `CertPath` object, and let the JDK's built-in Public Key Infrastructure (PKI) framework handle the validation. Internally, the CertPath implementation might create a `Signature` object, and use that to verify that the each of the signatures in the certificate chain.

With this basic understanding of the architecture, we can look at some of the steps in the SSL/TLS handshake. The client begins by sending a ClientHello message to the server. The server selects a ciphersuite to use, and sends that back in a ServerHello message, and begins creating JCA objects based on the suite selection. We'll use server-only authentication in the following examples.

Figure 2-18 SSL/TLS Messages



Server-only authentication is described in the following examples. The examples are vastly simplified, but gives an idea of how the JSSE classes might be combined to create a higher level protocol:

Example 2-9 SSL/TLS Server Uses a RSA-based ciphersuite Such as TLS_RSA_WITH_AES_128_CBC_SHA

```
KeyManagerTrustManagerSecureRandomCipherPublicKeyPrivateKeyCipher
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 2-10 Choose an Ephemeral Diffie-Hellman Key Agreement Algorithm Along with the DSA Signature Algorithm such as TLS_DHE_DSS_WITH_AES_128_CBC_SHA

`KeyPairGeneratorKeyFactoryDHPublicKeySpecKeyAgreementPrivateKeyServerKeyExchangeKeyFactoryKeyAgreement`
Once the actual encryption keys have been established, the secret key is used to initialize a symmetric `Cipher` object, and this cipher is used to protect all data in transit. To help determine if the data has been modified, a `MessageDigest` is created and receives a copy of the data destined for the network. When the packet is complete, the digest (hash) is appended to data, and the entire packet is encrypted by the `Cipher`. If a block cipher such as AES is used, the data must be padded to make a complete block. On the remote side, the steps are simply reversed.

Cryptographic Strength Configuration

You can configure the cryptographic strength of the Java Cryptography Extension (JCE) architecture using jurisdiction policy files (see Jurisdiction Policy File Format) and the security properties file.

Prior to Oracle Java JDK 9, the default cryptographic strength allowed by Oracle implementations was “strong but limited” (for example AES keys limited to 128 bits). To remove this restriction, administrators could download and install a separate “unlimited strength” Jurisdiction Policy Files bundle. The Jurisdiction Policy File mechanism was reworked for JDK 9. It now allows for much more flexible configuration. The Oracle JDK now ships with a default value of “unlimited” rather than “limited”. As always, administrators and users must still continue to follow all import/export guidelines for their geographical locations. The active cryptographic strength is now determined using a Security Property (typically set in the `java.security` properties file), in combination with the jurisdiction policy files found in the configuration directory.

All the necessary JCE policy files to provide either unlimited cryptographic strength or strong but limited cryptographic strength are bundled with the JDK.

Cryptographic Strength Settings

Each directory under `<java_home>/conf/security/policy` represents a set of policy configurations defined by the jurisdiction policy files that they contain. You activate a particular cryptographic strength setting represented by the policy files in a directory by setting the `crypto.policy` Security Property (configured in the file `<java_home>/conf/security/java.security`) to point to that directory.

> **Note**
>
> Properties in the `java.security` file are typically parsed only once. If you have modified any property in this file, restart your applications to ensure that the changes are properly reflected.

The JDK comes bundled with two such directories, `limited` and `unlimited`, each containing a number of policy files. By default, the `crypto.policy` Security Property is set to:

```
crypto.policy = unlimited
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The overall value is the intersection of the files contained within the directory. These policy files settings are VM-wide, and affect all applications running on this VM. If you want to override cryptographic strength at the application level, see How to Make Applications Exempt from Cryptographic Restrictions.

Unlimited Directory Contents

The `unlimited` directory contains the following policy files:

- `<java_home>/conf/security/unlimited/default_US_export.policy`

```
// Default US Export policy file.
grant {
// There is no restriction to any algorithms.
     permission javax.crypto.CryptoAllPermission;
};
```

> **Note**
>
> As there are no current restrictions on export of cryptography from the United States, the `default_US_export.policy` file is set with no restrictions.

- `<java_home>/conf/security/unlimited/default_local.policy`

```
// Country specific policy file for countries with no limits on crypto
strength.
grant {
// There is no restriction to any algorithms.
     permission javax.crypto.CryptoAllPermission;
};
```

> **Note**
>
> Depending on the country, there may be local restrictions, but as this policy file is located in the `unlimited` directory, there are no restrictions listed here.

To select unlimited cryptographic strength as defined in these two files set `crypto.policy = unlimited` in the file `<java_home>/conf/security/java.security`.

Limited Directory Contents

The `limited` directory currently contains the following policy files:

- `<java_home>/conf/security/limited/default_US_export.policy`

```
// Default US Export policy file.
grant {
// There is no restriction to any algorithms.
     permission javax.crypto.CryptoAllPermission;
};
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> Even though this is in the `limited` directory, as there are no current restrictions on export of cryptography from the United States, the `default_US_export.policy` file is set with no restrictions.

- `<java_home>/conf/security/limited/default_local.policy`

```
// Some countries have import limits on crypto strength. This policy file
// is worldwide importable.

grant {
    permission javax.crypto.CryptoPermission "DES", 64;
    permission javax.crypto.CryptoPermission "DESede", *;
    permission javax.crypto.CryptoPermission "RC2", 128,
                                        "javax.crypto.spec.RC2ParameterSpec",
128;
    permission javax.crypto.CryptoPermission "RC4", 128;
    permission javax.crypto.CryptoPermission "RC5", 128,
          "javax.crypto.spec.RC5ParameterSpec", *, 12, *;
    permission javax.crypto.CryptoPermission "RSA", *;
    permission javax.crypto.CryptoPermission *, 128;
};
```

> **Note**
>
> This local policy file shows the default restrictions. It should be allowed by any country, including those that have import restrictions, but please obtain legal guidance.

- `<java_home>/conf/security/limited/exempt_local.policy`

```
// Some countries have import limits on crypto strength, but may allow for
// these exemptions if the exemption mechanism is used.

grant {
    // There is no restriction to any algorithms if KeyRecovery is
enforced.
    permission javax.crypto.CryptoPermission *, "KeyRecovery";

    // There is no restriction to any algorithms if KeyEscrow is enforced.
    permission javax.crypto.CryptoPermission *, "KeyEscrow";

    // There is no restriction to any algorithms if KeyWeakening is
enforced.
    permission javax.crypto.CryptoPermission *, "KeyWeakening";
};
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> Countries that have import restrictions should use “limited”, but these restrictions could be relaxed if the exemption mechanism can be employed. See How to Make Applications Exempt from Cryptographic Restrictions. Please obtain legal guidance for your situation.

Custom Cryptographic Strength Settings

To set up restrictions to cryptographic strength that are different than the settings in the policy files in the `limited` or `unlimited` directory, you can create a new directory, parallel with `limited` and `unlimited`, and place your policy files there. For example, you may create a directory called `custom`. In this `custom` directory you include the files `default_*export.policy` and/or `exempt_*local.policy`.

To select cryptographic strength as defined in the files in the `custom` directory, set `crypto.policy = custom` in the file `<java_home>/conf/security/java.security`.

Jurisdiction Policy File Format

JCA represents its jurisdiction policy files as Java-style policy files with corresponding permission statements. As described in Cryptographic Strength Configuration, a Java policy file specifies what permissions are allowed for code from specified code sources. A permission represents access to a system resource. In the case of JCA, the "resources" are cryptography algorithms, and code sources do not need to be specified, because the cryptographic restrictions apply to all code.

A jurisdiction policy file consists of a very basic "grant entry" containing one or more "permission entries."

```
grant {
    <permission entries>;
};
```

The format of a permission entry in a jurisdiction policy file is:

```
permission <crypto permission class name>
    [<alg_name>
        [
            [, <exemption mechanism name>]
            [, <maxKeySize>
                [, <AlgorithmParameterSpec class name>,
                        <parameters for constructing an AlgorithmParameterSpec
object>
                ]
            ]
        ]
    ];
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

A sample jurisdiction policy file that includes restricting the AES algorithm to maximum key sizes of 128 bits is:

```
grant {
    permission javax.crypto.CryptoPermission "AES", 128;
    // ...
};
```

A permission entry must begin with the word `permission`. Items that appear in a permission entry must appear in the specified order. An entry is terminated with a semicolon. Case is unimportant for the identifiers (`grant`, `permission`) but is significant for the *<crypto permission class name>* or for any string that is passed in as a value. An asterisk (*) can be used as a wildcard for any permission entry option. For example, an asterisk for an *<alg_name>* option means "all algorithms."

The following table describes a permission entry's options:

Table 2-1 Permission Entry Options

Option	Description
<crypto permission class name>	Specific permission class name, such as `javax.crypto.CryptoPermission`. Required. A crypto permission class reflects the ability of an application to use certain algorithms with certain key sizes in certain environments. There are two crypto permission classes: `CryptoPermission` and `CryptoAllPermission`. The special `CryptoAllPermission` class implies all cryptography-related permissions, that is, it specifies that there are no cryptography-related restrictions.
<alg_name>	Quoted string specifying the standard name of a cryptography algorithm, such as "AES" or "RSA". Optional.
<exemption mechanism name>	Quoted string indicating an exemption mechanism which, if enforced, enables a reduction in cryptographic restrictions. Optional. Exemption mechanism names that can be used include "KeyRecovery" "KeyEscrow", and "KeyWeakening".
<maxKeySize>	Integer specifying the maximum key size (in bits) allowed for the specified algorithm. Optional.
<AlgorithmParameterSpec class name>	Class name that specifies the strength of the algorithm. Optional. For some algorithms, it may not be sufficient to specify the algorithm strength in terms of just a key size. For example, in the case of the "RC5" algorithm, the number of rounds must also be considered. For algorithms whose strength needs to be expressed as more than a key size, use this option to specify the `AlgorithmParameterSpec` class name that does this (such as `javax.crypto.spec.RC5ParameterSpec` for the "RC5" algorithm).

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 2-1 (Cont.) Permission Entry Options

Option	Description
<parameters for constructing an AlgorithmParameterSpec object>	List of parameters for constructing the specified `AlgorithmParameterSpec` object. Required if *<AlgorithmParameterSpec class name>* has been specified and requires parameters.

How to Make Applications Exempt from Cryptographic Restrictions

NOT_SUPPORTED

This section should be ignored by most application developers. It is only for people whose applications may be exported to those few countries whose governments mandate cryptographic restrictions, if it is desired that such applications have fewer cryptographic restrictions than those mandated.

By default, an application can use cryptographic algorithms of any strength. However, due to import control restrictions by the governments of a few countries, you may have to limit those algorithms' strength. The JCA framework includes an ability to enforce restrictions regarding the maximum strengths of cryptographic algorithms available to applications in different jurisdiction contexts (locations). You specify these restrictions in jurisdiction policy files. For more information about jurisdiction policy files and how to create and configure them, see Cryptographic Strength Configuration.

It is possible that the governments of some or all such countries may allow certain applications to become exempt from some or all cryptographic restrictions. For example, they may consider certain types of applications as "special" and thus exempt. Or they may exempt any application that utilizes an "exemption mechanism," such as key recovery. Applications deemed to be exempt could get access to stronger cryptography than that allowed for non-exempt applications in such countries.

In order for an application to be recognized as "exempt" at runtime, it must meet the following conditions:

- It must have a permission policy file bundled with it in a JAR file. The permission policy file specifies what cryptography-related permissions the application has, and under what conditions (if any).
- The JAR file containing the application and the permission policy file must have been signed using a code-signing certificate issued after the application was accepted as exempt.

The following are sample steps required in order to make an application exempt from some cryptographic restrictions. This is a basic outline that includes information about what is required by JCA in order to recognize and treat applications as being exempt. You will need to know the exemption requirements of the particular country or countries in which you would like your application to be able to be run but whose governments require cryptographic restrictions. You will also need to know the requirements of a JCA framework vendor that has a process in place for handling exempt applications. Consult such a vendor for further information.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The `SunJCE` provider does not supply an implementation of the `ExemptionMechanismSpi` class

1. Write and Compile Your Application Code
2. Create a Permission Policy File Granting Appropriate Cryptographic Permissions
3. Prepare for Testing
   a. Apply for Government Approval From the Government Mandating Restrictions.
   b. Get a Code-Signing Certificate
   c. Bundle the Application and Permission Policy File into a JAR file
   d. Step 7.1: Get a Code-Signing Certificate
   e. Set Up Your Environment Like That of a User in a Restricted Country
   f. (only for applications using exemption mechanisms) Install a Provider Implementing the Exemption Mechanism Specified by the entry in the Permission Policy File
4. Test Your Application
5. Apply for U.S. Government Export Approval If Required
6. Deploy Your Application

Special Code Requirements for Applications that Use Exemption Mechanisms

When an application has a permission policy file associated with it (in the same JAR file) and that permission policy file specifies an exemption mechanism, then when the `Cipher getInstance` method is called to instantiate a `Cipher`, the JCA code searches the installed providers for one that implements the specified exemption mechanism. If it finds such a provider, JCA instantiates an `ExemptionMechanism` API object associated with the provider's implementation, and then associates the `ExemptionMechanism` object with the `Cipher` returned by `getInstance`.

After instantiating a Cipher, and prior to initializing it (via a call to the Cipher `init` method), your code must call the following `Cipher` method:

```
public ExemptionMechanism getExemptionMechanism()
```

This call returns the `ExemptionMechanism` object associated with the Cipher. You must then initialize the exemption mechanism implementation by calling the following method on the returned `ExemptionMechanism`:

```
public final void init(Key key)
```

The argument you supply should be the same as the argument of the same types that you will subsequently supply to a Cipher `init` method.

Once you have initialized the `ExemptionMechanism`, you can proceed as usual to initialize and use the Cipher.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Permission Policy Files

In order for an application to be recognized at runtime as being "exempt" from some or all cryptographic restrictions, it must have a permission policy file bundled with it in a JAR file. The permission policy file specifies what cryptography-related permissions the application has, and under what conditions (if any).

The format of a permission entry in a permission policy file that accompanies an exempt application is the same as the format for a jurisdiction policy file downloaded with the JDK, which is:

```
permission <crypto permission class name>
    [<alg_name>
        [
            [, <exemption mechanism name>]
            [, <maxKeySize>
                [, <AlgorithmParameterSpec class name>,
                       <parameters for constructing an AlgorithmParameterSpec
object>
                ]
            ]
        ]
    ];
```

See Jurisdiction Policy File Format.

Permission Policy Files for Exempt Applications

Some applications may be allowed to be completely unrestricted. Thus, the permission policy file that accompanies such an application usually just needs to contain the following:

```
grant {
    // There are no restrictions to any algorithms.
    permission javax.crypto.CryptoAllPermission;
};
```

If an application just uses a single algorithm (or several specific algorithms), then the permission policy file could simply mention that algorithm (or algorithms) explicitly, rather than granting `CryptoAllPermission`.

For example, if an application just uses the Blowfish algorithm, the permission policy file doesn't have to grant `CryptoAllPermission` to all algorithms. It could just specify that there is no cryptographic restriction if the Blowfish algorithm is used. In order to do this, the permission policy file would look like the following:

```
grant {
    permission javax.crypto.CryptoPermission "Blowfish";
};
```

Permission Policy Files for Applications Exempt Due to Exemption Mechanisms

If an application is considered "exempt" if an exemption mechanism is enforced, then the permission policy file that accompanies the application must specify one or more exemption mechanisms. At run time, the application will be considered exempt if any of those exemption

Copyright © 1993, 2026, Oracle and/or its affiliates.

mechanisms is enforced. Each exemption mechanism must be specified in a permission entry that looks like the following:

```
// No algorithm restrictions if specified
// exemption mechanism is enforced.
permission javax.crypto.CryptoPermission *,
    "<ExemptionMechanismName>";
```

where <*ExemptionMechanismName*> specifies the name of an exemption mechanism. The list of possible exemption mechanism names includes:

- `KeyRecovery`
- `KeyEscrow`
- `KeyWeakening`

As an example, suppose your application is exempt if either key recovery or key escrow is enforced. Then your permission policy file should contain the following:

```
grant {
    // No algorithm restrictions if KeyRecovery is enforced.
    permission javax.crypto.CryptoPermission *, "KeyRecovery";

    // No algorithm restrictions if KeyEscrow is enforced.
    permission javax.crypto.CryptoPermission *, "KeyEscrow";
};
```

> **Note**
>
> Permission entries that specify exemption mechanisms should *not* also specify maximum key sizes. The allowed key sizes are actually determined from the installed exempt jurisdiction policy files, as described in the next section.

How Bundled Permission Policy Files Affect Cryptographic Permissions

At runtime, when an application instantiates a `Cipher` (via a call to its `getInstance` method) and that application has an associated permission policy file, JCA checks to see whether the permission policy file has an entry that applies to the algorithm specified in the `getInstance` call. If it does, and the entry grants `CryptoAllPermission` or does not specify that an exemption mechanism must be enforced, it means there is no cryptographic restriction for this particular algorithm.

If the permission policy file has an entry that applies to the algorithm specified in the `getInstance` call and the entry *does* specify that an exemption mechanism must be enforced, then the exempt jurisdiction policy file(s) are examined. If the exempt permissions include an entry for the relevant algorithm and exemption mechanism, and that entry is implied by the permissions in the permission policy file bundled with the application, and if there is an implementation of the specified exemption mechanism available from one of the registered providers, then the maximum key size and algorithm parameter values for the `Cipher` are determined from the exempt permission entry.

If there is no exempt permission entry implied by the relevant entry in the permission policy file bundled with the application, or if there is no implementation of the specified exemption mechanism available from any of the registered providers, then the application is only allowed the standard default cryptographic permissions.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Packaging Your Application

You can package an application in three different kinds of modules:

- Named or explicit module: A module that appears on the module path and contains module configuration information in the `module-info.class` file.
- Automatic module: A module that appears on the module path, but does not contain module configuration information in a `module-info.class` file (essentially a "regular" JAR file).
- Unnamed module: A module that appears on the class path. It may or may not have a `module-info.class` file; this file is ignored.

It is recommended that you package your applications in named modules as they provide better performance, stronger encapsulation, and simpler configuration. They also offer greater flexibility; you can use them with non-modular JDKs or even as unnamed modules by specifying them in a modular JDK's class path.

For more information about modules, see The State of the Module System and JEP 261: Module System

Additional JCA Code Samples

These examples illustrate use of several JCA mechanisms. See also Sample Programs for Diffie-Hellman Key Exchange, AES/GCM, HMAC-SHA256, and HPKE

Topics

Computing a MessageDigest Object

Generating a Pair of Keys

Generating and Verifying a Signature Using Generated Keys

Generating/Verifying Signatures Using Key Specifications and KeyFactory

Generating Random Numbers

Determining If Two Keys Are Equal

Reading Base64-Encoded Certificates

Parsing a Certificate Reply

Using Encryption

Using Password-Based Encryption

Encapsulating and Decapsulating Keys

Computing a MessageDigest Object

These steps describe the procedure to compute a `MessageDigest` object.

Copyright © 1993, 2026, Oracle and/or its affiliates.

1. Create the `MessageDigest` object, as in the following example:

   ```
   MessageDigest sha = MessageDigest.getInstance("SHA-256");
   ```

   This call assigns a properly initialized message digest object to the `sha` variable. The implementation implements the Secure Hash Algorithm (SHA-256), as defined in the National Institute for Standards and Technology's (NIST) FIPS 180-4 document.

2. Suppose we have three byte arrays, `i1`, `i2` and `i3`, which form the total input whose message digest we want to compute. This digest (or "hash") could be calculated via the following calls:

   ```
   sha.update(i1);
   sha.update(i2);
   sha.update(i3);
   byte[] hash = sha.digest();
   ```

3. Optional: An equivalent alternative series of calls would be:

   ```
   sha.update(i1);
   sha.update(i2);
   byte[] hash = sha.digest(i3);
   ```

   After the message digest has been calculated, the message digest object is automatically reset and ready to receive new data and calculate its digest. All former state (i.e., the data supplied to `update` calls) is lost.

Example 2-11 Hash Implementations Through Cloning

Some hash implementations may support intermediate hashes through cloning. Suppose we want to calculate separate hashes for:

- `i1`
- `i1 and i2`
- `i1, i2, and i3`

The following is one way to calculate these hashes; however, this code works only if the SHA-256 implementation is cloneable:

```
/* compute the hash for i1 */
sha.update(i1);
byte[] i1Hash = sha.clone().digest();

/* compute the hash for i1 and i2 */
sha.update(i2);
byte[] i12Hash = sha.clone().digest();

/* compute the hash for i1, i2 and i3 */
sha.update(i3);
byte[] i123hash = sha.digest();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 2-12 Determine if the Hash Implementation is Cloneable or not Cloneable

```
MessageDigest

try {
    // try and clone it
    /* compute the hash for i1 */
    sha.update(i1);
    byte[] i1Hash = sha.clone().digest();
    // ...
    byte[] i123hash = sha.digest();
} catch (CloneNotSupportedException cnse) {
    // do something else, such as the code in the section
    // "Compute Intermediate Digests if the Hash Implementation is not
Cloneable"
}
```

Example 2-13 Compute Intermediate Digests if the Hash Implementation is not Cloneable

```
MessageDigest md1 = MessageDigest.getInstance("SHA-256");
MessageDigest md2 = MessageDigest.getInstance("SHA-256");
MessageDigest md3 = MessageDigest.getInstance("SHA-256");

byte[] i1Hash = md1.digest(i1);

md2.update(i1);
byte[] i12Hash = md2.digest(i2);

md3.update(i1);
md3.update(i2);
byte[] i123Hash = md3.digest(i3);
```

Generating a Pair of Keys

In this example we will generate a public-private key pair for the algorithm named "DSA" (Digital Signature Algorithm), and use this keypair in future examples. We will generate keys with a 2048-bit modulus. We don't care which provider supplies the algorithm implementation.

Creating the Key Pair Generator

The first step is to get a key pair generator object for generating keys for the DSA algorithm:

```
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
```

Initializing the Key Pair Generator

The next step is to initialize the key pair generator. In most cases, algorithm-independent initialization is sufficient, but in some cases, algorithm-specific initialization is used.

Algorithm-Independent Initialization

All key pair generators share the concepts of a keysize and a source of randomness. The `KeyPairGenerator` class initialization methods at a minimum needs a keysize. If the source of

Copyright © 1993, 2026, Oracle and/or its affiliates.

randomness is not explicitly provided, a `SecureRandom` implementation of the highest-priority installed provider will be used. Thus to generate keys with a keysize of 2048, simply call:

```
keyGen.initialize(2048);
```

The following code illustrates how to use a specific, additionally seeded `SecureRandom` object:

```
SecureRandom random = SecureRandom.getInstance("DRBG", "SUN");
random.setSeed(userSeed);
keyGen.initialize(2048, random);
```

Since no other parameters are specified when you call these algorithm-independent `initialize` method, it is up to the provider what to do about the algorithm-specific parameters (if any) to be associated with each of the keys. The provider may use precomputed parameter values or may generate new values.

Algorithm-Specific Initialization

For situations where a set of algorithm-specific parameters already exists (such as "community parameters" in DSA), there are two `initialize` methods that have an `AlgorithmParameterSpec` argument. Suppose your key pair generator is for the "DSA" algorithm, and you have a set of DSA-specific parameters, `p`, `q`, and `g`, that you would like to use to generate your key pair. You could execute the following code to initialize your key pair generator (recall that `DSAParameterSpec` is an `AlgorithmParameterSpec`):

```
DSAParameterSpec dsaSpec = new DSAParameterSpec(p, q, g);
keyGen.initialize(dsaSpec);
```

Generating the Pair of Keys

The final step is actually generating the key pair. No matter which type of initialization was used (algorithm-independent or algorithm-specific), the same code is used to generate the `KeyPair`:

```
KeyPair pair = keyGen.generateKeyPair();
```

Generating and Verifying a Signature Using Generated Keys

Examples of generating and verifying a signature using generated keys.

The following signature generation and verification examples use the `KeyPair` generated in the Generating a Pair of Keys .

Generating a Signature

We first create a Signature Class object:

```
Signature dsa = Signature.getInstance("SHA256withDSA");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Next, using the key pair generated in the key pair example, we initialize the object with the private key, then sign a byte array called `data`.

```
/* Initializing the object with a private key */
 PrivateKey priv = pair.getPrivate();
 dsa.initSign(priv);

/* Update and sign the data */
 dsa.update(data);
 byte[] sig = dsa.sign();
```

Verifying a Signature

Verifying the signature is straightforward. (Note that here we also use the key pair generated in the key pair example.)

```
/* Initializing the object with the public key */
PublicKey pub = pair.getPublic();
dsa.initVerify(pub);

/* Update and verify the data */
dsa.update(data);
boolean verifies = dsa.verify(sig);
System.out.println("signature verifies: " + verifies);
```

Generating/Verifying Signatures Using Key Specifications and KeyFactory

Suppose that, rather than having a public/private key pair (as, for example, was generated in the section Generating a Pair of Keys), you simply have the components of your DSA private key: `x` (the private key), `p` (the prime), `q` (the sub-prime), and `g` (the base).

Furthermore, suppose you want to use your private key to digitally sign some data, which is in a byte array named `someData`. You would do the following steps, which also illustrate creating a key specification and using a key factory to obtain a `PrivateKey` from the key specification (`initSign` requires a `PrivateKey`):

```
DSAPrivateKeySpec dsaPrivKeySpec = new DSAPrivateKeySpec(x, p, q, g);

KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PrivateKey privKey = keyFactory.generatePrivate(dsaPrivKeySpec);

Signature sig = Signature.getInstance("SHA256withDSA");
sig.initSign(privKey);
sig.update(someData);
byte[] signature = sig.sign();
```

Suppose Alice wants to use the data you signed. In order for her to do so, and to verify your signature, you need to send her three things:

1. The data
2. The signature

Copyright © 1993, 2026, Oracle and/or its affiliates.

3. The public key corresponding to the private key you used to sign the data

You can store the `someData` bytes in one file, and the `signature` bytes in another, and send those to Alice.

For the public key, assume, as in the previous signing example, you have the components of the DSA public key corresponding to the DSA private key used to sign the data. Then you can create a `DSAPublicKeySpec` from those components:

```
DSAPublicKeySpec dsaPubKeySpec = new DSAPublicKeySpec(y, p, q, g);
```

You still need to extract the key bytes so that you can put them in a file. To do so, you can first call the `generatePublic` method on the DSA key factory already created in the previous example:

```
PublicKey pubKey = keyFactory.generatePublic(dsaPubKeySpec);
```

Then you can extract the (encoded) key bytes via the following:

```
byte[] encKey = pubKey.getEncoded();
```

You can now store these bytes in a file, and send it to Alice along with the files containing the data and the signature.

Now, assume Alice has received these files, and she copied the data bytes from the data file to a byte array named `data`, the signature bytes from the signature file to a byte array named `signature`, and the encoded public key bytes from the public key file to a byte array named `encodedPubKey`.

Alice can now execute the following code to verify the signature. The code also illustrates how to use a key factory in order to instantiate a DSA public key from its encoding (`initVerify` requires a `PublicKey`).

```
X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encodedPubKey);

KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

Signature sig = Signature.getInstance("SHA256withDSA");
sig.initVerify(pubKey);
sig.update(data);
sig.verify(signature);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> In the previous example, Alice needed to generate a `PublicKey` from the encoded key bits, since `initVerify` requires a `PublicKey` . Once she has a `PublicKey`, she could also use the `KeyFactorygetKeySpec` method to convert it to a `DSAPublicKeySpec` so that she can access the components, if desired, as in:

```
    DSAPublicKeySpec dsaPubKeySpec =
        (DSAPublicKeySpec)keyFactory.getKeySpec(pubKey,
DSAPublicKeySpec.class);
```

Now she can access the DSA public key components `y`, `p`, `q`, and `g` through the corresponding "get" methods on the `DSAPublicKeySpec` class (`getY`, `getP`, `getQ`, and `getG`).

Generating Random Numbers

The following code sample illustrates generating random numbers configured with different security strengths using a DRBG implementation of the `SecureRandom` class:

```
    SecureRandom drbg;
    byte[] buffer = new byte[32];

    // Any DRBG can be provided
    drbg = SecureRandom.getInstance("DRBG");
    drbg.nextBytes(buffer);

    SecureRandomParameters params = drbg.getParameters();
    if (params instanceof DrbgParameters.Instantiation) {
        DrbgParameters.Instantiation ins = (DrbgParameters.Instantiation)
params;
        if (ins.getCapability().supportsReseeding()) {
            drbg.reseed();
        }
    }

    // The following call requests a weak DRBG instance. It is only
    // guaranteed to support 112 bits of security strength.
    drbg = SecureRandom.getInstance("DRBG",
        DrbgParameters.instantiation(112, NONE, null));

    // Both the next two calls will likely fail, because drbg could be
    // instantiated with a smaller strength with no prediction resistance
    // support.
    drbg.nextBytes(buffer,
        DrbgParameters.nextBytes(256, false, "more".getBytes()));
    drbg.nextBytes(buffer,
        DrbgParameters.nextBytes(112, true, "more".getBytes()));

    // The following call requests a strong DRBG instance, with a
    // personalization string. If it successfully returns an instance,
    // that instance is guaranteed to support 256 bits of security strength
    // with prediction resistance available.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
drbg = SecureRandom.getInstance("DRBG", DrbgParameters.instantiation(
    256, PR_AND_RESEED, "hello".getBytes()));

// Prediction resistance is not requested in this single call,
// but an additional input is used.
drbg.nextBytes(buffer,
    DrbgParameters.nextBytes(-1, false, "more".getBytes()));

// Same for this call.
drbg.reseed(DrbgParameters.reseed(false, "extra".getBytes()));
```

Determining If Two Keys Are Equal

Example code for determining if two keys are equal.

In many cases you would like to know if two keys are equal; however, the default method `java.lang.Object.equals` may not give the desired result. The most provider-independent approach is to compare the encoded keys. If this comparison isn't appropriate (for example, when comparing an `RSAPrivateKey` and an `RSAPrivateCrtKey`), you should compare each component.

The following code demonstrates this idea:

```
static boolean keysEqual(Key key1, Key key2) {
    if (key1.equals(key2)) {
       return true;
    }

    if (Arrays.equals(key1.getEncoded(), key2.getEncoded())) {
       return true;
    }

 // More code for different types of keys here.
 // For example, the following code can check if
 // an RSAPrivateKey and an RSAPrivateCrtKey are equal:
 // if ((key1 instanceof RSAPrivateKey) &&
 //     (key2 instanceof RSAPrivateKey)) {
 //     if ((key1.getModulus().equals(key2.getModulus())) &&
 //         (key1.getPrivateExponent().equals(
 //                                       key2.getPrivateExponent()))) {
 //         return true;
 //     }
 // }

     return false;
 }
```

Reading Base64-Encoded Certificates

The following example reads a file with Base64-encoded certificates, which are each bounded at the beginning by

```
-----BEGIN CERTIFICATE-----
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

and at the end by

```
-----END CERTIFICATE-----
```

We convert the `FileInputStream` (which does not support `mark` and `reset`) to a `ByteArrayInputStream` (which supports those methods), so that each call to `generateCertificate` consumes only one certificate, and the read position of the input stream is positioned to the next certificate in the file:

```
 try (FileInputStream fis = new FileInputStream(filename);
     BufferedInputStream bis = new BufferedInputStream(fis)) {
     CertificateFactory cf = CertificateFactory.getInstance("X.509");
     while (bis.available() > 0) {
         Certificate cert = cf.generateCertificate(bis);
         System.out.println(cert.toString());
     }
}
```

Parsing a Certificate Reply

The following example parses a PKCS7-formatted certificate reply stored in a file and extracts all the certificates from it:

```
try (FileInputStream fis = new FileInputStream(filename)) {
   CertificateFactory cf = CertificateFactory.getInstance("X.509");

   Collection<? extends Certificate> c = cf.generateCertificates(fis);
   for (Certificate cert : c) {
       System.out.println(cert);
   }

   // Alternatively, use this aggregate operation instead of a for-loop:
   // c.stream().forEach(e -> System.out.println(e));
}
```

Using Encryption

This section takes the user through the process of generating a key, creating and initializing a cipher object, encrypting a file, and then decrypting it. Throughout this example, we use the Advanced Encryption Standard (AES).

Generating a Key

To create an AES key, we have to instantiate a `KeyGenerator` for AES. We do not specify a provider, because we do not care about a particular AES key generation implementation. Since we do not initialize the `KeyGenerator`, a system-provided source of randomness and a default keysize will be used to create the AES key:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecretKey aesKey = keygen.generateKey();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

After the key has been generated, the same KeyGenerator object can be re-used to create further keys.

Creating a Cipher

The next step is to create a `Cipher` instance. To do this, we use one of the `getInstance` factory methods of the `Cipher` class. We must specify the name of the requested transformation, which includes the following components, separated by slashes (/):

- the algorithm name
- the mode (optional)
- the padding scheme (optional)

In this example, we create an AES cipher in Cipher Block Chaining mode, with PKCS5-style padding. We do not specify a provider, because we do not care about a particular implementation of the requested transformation.

The standard algorithm name for AES is "AES", the standard name for the Cipher Block Chaining mode is "CBC", and the standard name for PKCS5-style padding is "PKCS5Padding":

```
Cipher aesCipher;

// Create the cipher
aesCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

We use the `aesKey` generated previously to initialize the `Cipher` object for encryption:

```
// Initialize the cipher for encryption
aesCipher.init(Cipher.ENCRYPT_MODE, aesKey);

// Our cleartext
byte[] cleartext = "This is just an example".getBytes();

// Encrypt the cleartext
byte[] ciphertext = aesCipher.doFinal(cleartext);

// Retrieve the parameters used during encryption to properly
// initialize the cipher for decryption
AlgorithmParameters params = aesCipher.getParameters();

// Initialize the same cipher for decryption
aesCipher.init(Cipher.DECRYPT_MODE, aesKey, params);

// Decrypt the ciphertext
byte[] cleartext1 = aesCipher.doFinal(ciphertext);
```

`cleartext` and `cleartext1` are identical.

Using Password-Based Encryption

In this example, we prompt the user for a password from which we derive an encryption key.

It would seem logical to collect and store the password in an object of type `java.lang.String`. However, here's the caveat: Objects of type `String` are immutable, i.e., there are no methods

Copyright © 1993, 2026, Oracle and/or its affiliates.

defined that allow you to change (overwrite) or zero out the contents of a `String` after usage. This feature makes `String` objects unsuitable for storing security sensitive information such as user passwords. You should always collect and store security sensitive information in a char array instead. For that reason, the `javax.crypto.spec.PBEKeySpec` class takes (and returns) a password as a char array.

In order to use Password-Based Encryption (PBE) as defined in PKCS5, we have to specify a *salt* and an *iteration count*. The same salt and iteration count that are used for encryption must be used for decryption. Newer PBE algorithms use an iteration count of at least 1000.

```
    PBEKeySpec pbeKeySpec;
    PBEParameterSpec pbeParamSpec;
    SecretKeyFactory keyFac;

    // Salt
    byte[] salt = new SecureRandom().nextBytes(salt);

    // Iteration count
    int count = 1000;

    // Create PBE parameter set
    pbeParamSpec = new PBEParameterSpec(salt, count);

    // Prompt user for encryption password.
    // Collect user password as char array, and convert
    // it into a SecretKey object, using a PBE key
    // factory.
    char[] password = System.console.readPassword("Enter encryption password:
");
    pbeKeySpec = new PBEKeySpec(password);
    keyFac = SecretKeyFactory.getInstance("PBEWithHmacSHA256AndAES_256");
    SecretKey pbeKey = keyFac.generateSecret(pbeKeySpec);

    // Create PBE Cipher
    Cipher pbeCipher = Cipher.getInstance("PBEWithHmacSHA256AndAES_256");

    // Initialize PBE Cipher with key and parameters
    pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParamSpec);

    // Our cleartext
    byte[] cleartext = "This is another example".getBytes();

    // Encrypt the cleartext
    byte[] ciphertext = pbeCipher.doFinal(cleartext);
```

Encapsulating and Decapsulating Keys

See The KEM Class for more information about key encapsulation and decapsulation.

```
    // Receiver side
    var kpg = KeyPairGenerator.getInstance("X25519");
    var kp = kpg.generateKeyPair();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
// Sender side
var kem1 = KEM.getInstance("DHKEM");
var sender = kem1.newEncapsulator(kp.getPublic());
var encapsulated = sender.encapsulate();
var k1 = encapsulated.key();

// Receiver side
var kem2 = KEM.getInstance("DHKEM");
var receiver = kem2.newDecapsulator(kp.getPrivate());
var k2 = receiver.decapsulate(encapsulated.encapsulation());

assert Arrays.equals(k1.getEncoded(), k2.getEncoded());
```

Sample Programs for Diffie-Hellman Key Exchange, AES/GCM, HMAC-SHA256, and HPKE

The following are sample programs for Diffie-Hellman key exchange, AES/GCM, and HMAC-SHA256.

Topics

Diffie-Hellman Key Exchange between Two Parties

Diffie-Hellman Key Exchange between Three Parties

AES/GCM Example

HMAC-SHA256 Example

Securely Exchanging Messages with HPKE and the HPKEParameterSpec Class

Diffie-Hellman Key Exchange between Two Parties

The program runs the Diffie-Hellman key agreement protocol between two parties.

```
import java.io.*;
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;
import java.security.interfaces.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;
import com.sun.crypto.provider.SunJCE;

public class DHKeyAgreement2 {
    private DHKeyAgreement2() {}
    public static void main(String argv[]) throws Exception {

        /*
         * Alice creates her own DH key pair with 2048-bit key size
         */
        System.out.println("ALICE: Generate DH keypair ...");
        KeyPairGenerator aliceKpairGen = KeyPairGenerator.getInstance("DH");
        aliceKpairGen.initialize(2048);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            KeyPair aliceKpair = aliceKpairGen.generateKeyPair();

            // Alice creates and initializes her DH KeyAgreement object
            System.out.println("ALICE: Initialization ...");
            KeyAgreement aliceKeyAgree = KeyAgreement.getInstance("DH");
            aliceKeyAgree.init(aliceKpair.getPrivate());

            // Alice encodes her public key, and sends it over to Bob.
            byte[] alicePubKeyEnc = aliceKpair.getPublic().getEncoded();

            /*
             * Let's turn over to Bob. Bob has received Alice's public key
             * in encoded format.
             * He instantiates a DH public key from the encoded key material.
             */
            KeyFactory bobKeyFac = KeyFactory.getInstance("DH");
            X509EncodedKeySpec x509KeySpec = new
X509EncodedKeySpec(alicePubKeyEnc);

            PublicKey alicePubKey = bobKeyFac.generatePublic(x509KeySpec);

            /*
             * Bob gets the DH parameters associated with Alice's public key.
             * He must use the same parameters when he generates his own key
             * pair.
             */
            DHParameterSpec dhParamFromAlicePubKey =
((DHPublicKey)alicePubKey).getParams();

            // Bob creates his own DH key pair
            System.out.println("BOB: Generate DH keypair ...");
            KeyPairGenerator bobKpairGen = KeyPairGenerator.getInstance("DH");
            bobKpairGen.initialize(dhParamFromAlicePubKey);
            KeyPair bobKpair = bobKpairGen.generateKeyPair();

            // Bob creates and initializes his DH KeyAgreement object
            System.out.println("BOB: Initialization ...");
            KeyAgreement bobKeyAgree = KeyAgreement.getInstance("DH");
            bobKeyAgree.init(bobKpair.getPrivate());

            // Bob encodes his public key, and sends it over to Alice.
            byte[] bobPubKeyEnc = bobKpair.getPublic().getEncoded();

            /*
             * Alice uses Bob's public key for the first (and only) phase
             * of her version of the DH
             * protocol.
             * Before she can do so, she has to instantiate a DH public key
             * from Bob's encoded key material.
             */
            KeyFactory aliceKeyFac = KeyFactory.getInstance("DH");
            x509KeySpec = new X509EncodedKeySpec(bobPubKeyEnc);
            PublicKey bobPubKey = aliceKeyFac.generatePublic(x509KeySpec);
            System.out.println("ALICE: Execute PHASE1 ...");
            aliceKeyAgree.doPhase(bobPubKey, true);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            /*
             * Bob uses Alice's public key for the first (and only) phase
             * of his version of the DH
             * protocol.
             */
            System.out.println("BOB: Execute PHASE1 ...");
            bobKeyAgree.doPhase(alicePubKey, true);

            /*
             * At this stage, both Alice and Bob have completed the DH key
             * agreement protocol.
             * Both generate the (same) shared secret.
             */
            try {
                byte[] aliceSharedSecret = aliceKeyAgree.generateSecret();
                int aliceLen = aliceSharedSecret.length;
                byte[] bobSharedSecret = new byte[aliceLen];
                int bobLen;
            } catch (ShortBufferException e) {
                System.out.println(e.getMessage());
            }        // provide output buffer of required size
            bobLen = bobKeyAgree.generateSecret(bobSharedSecret, 0);
            System.out.println("Alice secret: " +
                    toHexString(aliceSharedSecret));
            System.out.println("Bob secret: " +
                    toHexString(bobSharedSecret));
            if (!java.util.Arrays.equals(aliceSharedSecret, bobSharedSecret))
                throw new Exception("Shared secrets differ");
            System.out.println("Shared secrets are the same");

            /*
             * Now let's create a SecretKey object using the shared secret
             * and use it for encryption. First, we generate SecretKeys for the
             * "AES" algorithm (based on the raw shared secret data) and
             * Then we use AES in CBC mode, which requires an initialization
             * vector (IV) parameter. Note that you have to use the same IV
             * for encryption and decryption: If you use a different IV for
             * decryption than you used for encryption, decryption will fail.
             *
             * If you do not specify an IV when you initialize the Cipher
             * object for encryption, the underlying implementation will generate
             * a random one, which you have to retrieve using the
             * javax.crypto.Cipher.getParameters() method, which returns an
             * instance of java.security.AlgorithmParameters. You need to transfer
             * the contents of that object (e.g., in encoded format, obtained via
             * the AlgorithmParameters.getEncoded() method) to the party who will
             * do the decryption. When initializing the Cipher for decryption,
             * the (reinstantiated) AlgorithmParameters object must be explicitly
             * passed to the Cipher.init() method.
             */
            System.out.println("Use shared secret as SecretKey object ...");
            SecretKeySpec bobAesKey = new SecretKeySpec(bobSharedSecret, 0, 16,
"AES");
            SecretKeySpec aliceAesKey = new SecretKeySpec(aliceSharedSecret, 0,
16, "AES");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            /*
             * Bob encrypts, using AES in CBC mode
             */
            Cipher bobCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            bobCipher.init(Cipher.ENCRYPT_MODE, bobAesKey);
            byte[] cleartext = "This is just an example".getBytes();
            byte[] ciphertext = bobCipher.doFinal(cleartext);

            // Retrieve the parameter that was used, and transfer it to Alice in
            // encoded format
            byte[] encodedParams = bobCipher.getParameters().getEncoded();

            /*
             * Alice decrypts, using AES in CBC mode
             */

            // Instantiate AlgorithmParameters object from parameter encoding
            // obtained from Bob
            AlgorithmParameters aesParams =
AlgorithmParameters.getInstance("AES");
            aesParams.init(encodedParams);
            Cipher aliceCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            aliceCipher.init(Cipher.DECRYPT_MODE, aliceAesKey, aesParams);
            byte[] recovered = aliceCipher.doFinal(ciphertext);
            if (!java.util.Arrays.equals(cleartext, recovered))
                throw new Exception("AES in CBC mode recovered text is " +
                        "different from cleartext");
            System.out.println("AES in CBC mode recovered text is "
                    "same as cleartext");
        }

        /*
         * Converts a byte to hex digit and writes to the supplied buffer
         */
        private static void byte2hex(byte b, StringBuffer buf) {
            char[] hexChars = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
                    '9', 'A', 'B', 'C', 'D', 'E', 'F' };
            int high = ((b & 0xf0) >> 4);
            int low = (b & 0x0f);
            buf.append(hexChars[high]);
            buf.append(hexChars[low]);
        }

        /*
         * Converts a byte array to hex string
         */
        private static String toHexString(byte[] block) {
            StringBuffer buf = new StringBuffer();
            int len = block.length;
            for (int i = 0; i < len; i++) {
                byte2hex(block[i], buf);
                if (i < len-1) {
                    buf.append(":");
                }
            }
            return buf.toString();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    }
}
```

Diffie-Hellman Key Exchange between Three Parties

The program runs the Diffie-Hellman key agreement protocol between 3 parties.

```
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;
   /*
    * This program executes the Diffie-Hellman key agreement protocol between
    * 3 parties: Alice, Bob, and Carol using a shared 2048-bit DH parameter.
    */
    public class DHKeyAgreement3 {
        private DHKeyAgreement3() {}
        public static void main(String argv[]) throws Exception {
        // Alice creates her own DH key pair with 2048-bit key size
            System.out.println("ALICE: Generate DH keypair ...");
            KeyPairGenerator aliceKpairGen =
KeyPairGenerator.getInstance("DH");
            aliceKpairGen.initialize(2048);
            KeyPair aliceKpair = aliceKpairGen.generateKeyPair();
        // This DH parameters can also be constructed by creating a
        // DHParameterSpec object using agreed-upon values
            DHParameterSpec dhParamShared =
((DHPublicKey)aliceKpair.getPublic()).getParams();
        // Bob creates his own DH key pair using the same params
            System.out.println("BOB: Generate DH keypair ...");
            KeyPairGenerator bobKpairGen = KeyPairGenerator.getInstance("DH");
            bobKpairGen.initialize(dhParamShared);
            KeyPair bobKpair = bobKpairGen.generateKeyPair();
        // Carol creates her own DH key pair using the same params
            System.out.println("CAROL: Generate DH keypair ...");
            KeyPairGenerator carolKpairGen =
KeyPairGenerator.getInstance("DH");
            carolKpairGen.initialize(dhParamShared);
            KeyPair carolKpair = carolKpairGen.generateKeyPair();
        // Alice initialize
            System.out.println("ALICE: Initialize ...");
            KeyAgreement aliceKeyAgree = KeyAgreement.getInstance("DH");
            aliceKeyAgree.init(aliceKpair.getPrivate());
        // Bob initialize
            System.out.println("BOB: Initialize ...");
            KeyAgreement bobKeyAgree = KeyAgreement.getInstance("DH");
            bobKeyAgree.init(bobKpair.getPrivate());
        // Carol initialize
            System.out.println("CAROL: Initialize ...");
            KeyAgreement carolKeyAgree = KeyAgreement.getInstance("DH");
            carolKeyAgree.init(carolKpair.getPrivate());
        // Alice uses Carol's public key
            Key ac = aliceKeyAgree.doPhase(carolKpair.getPublic(), false);
        // Bob uses Alice's public key
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                Key ba = bobKeyAgree.doPhase(aliceKpair.getPublic(), false);
            // Carol uses Bob's public key
                Key cb = carolKeyAgree.doPhase(bobKpair.getPublic(), false);
            // Alice uses Carol's result, cb
                aliceKeyAgree.doPhase(cb, true);
            // Bob uses Alice's result, ac
                bobKeyAgree.doPhase(ac, true);
            // Carol uses Bob's result, ba
                carolKeyAgree.doPhase(ba, true);
            // Alice, Bob and Carol compute their secrets
                byte[] aliceSharedSecret = aliceKeyAgree.generateSecret();
                System.out.println("Alice secret: " +
toHexString(aliceSharedSecret));
                byte[] bobSharedSecret = bobKeyAgree.generateSecret();
                System.out.println("Bob secret: " + toHexString(bobSharedSecret));
                byte[] carolSharedSecret = carolKeyAgree.generateSecret();
                System.out.println("Carol secret: " +
toHexString(carolSharedSecret));
            // Compare Alice and Bob
                if (!java.util.Arrays.equals(aliceSharedSecret, bobSharedSecret))
                    throw new Exception("Alice and Bob differ");
                System.out.println("Alice and Bob are the same");
            // Compare Bob and Carol
                if (!java.util.Arrays.equals(bobSharedSecret, carolSharedSecret))
                    throw new Exception("Bob and Carol differ");
                System.out.println("Bob and Carol are the same");
            }
        /*
         * Converts a byte to hex digit and writes to the supplied buffer
         */
            private static void byte2hex(byte b, StringBuffer buf) {
                char[] hexChars = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
                                    '9', 'A', 'B', 'C', 'D', 'E', 'F' };
                int high = ((b & 0xf0) >> 4);
                int low = (b & 0x0f);
                buf.append(hexChars[high]);
                buf.append(hexChars[low]);
            }
        /*
         * Converts a byte array to hex string
         */
            private static String toHexString(byte[] block) {
                StringBuffer buf = new StringBuffer();
                int len = block.length;
                for (int i = 0; i < len; i++) {
                    byte2hex(block[i], buf);
                    if (i < len-1) {
                        buf.append(":");
                    }
                }
                return buf.toString();
            }
        }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

AES/GCM Example

The following is a sample program to demonstrate AES/GCM usage to encrypt/decrypt data.

```
import java.security.AlgorithmParameters;
import java.util.Arrays;
import javax.crypto.*;

public class AESGCMTest {

    public static void main(String[] args) throws Exception {
        // Slightly longer than 1 AES block (128 bits) to show PADDING
        // is "handled" by GCM.
        byte[] data = {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
            0x10};

        // Create a 128-bit AES key.
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        kg.init(128);
        SecretKey key = kg.generateKey();

        // Obtain a AES/GCM cipher to do the enciphering. Must obtain
        // and use the Parameters for successful decryption.
        Cipher encCipher = Cipher.getInstance("AES/GCM/NOPADDING");
        encCipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] enc = encCipher.doFinal(data);
        AlgorithmParameters ap = encCipher.getParameters();

        // Obtain a similar cipher, and use the parameters.
        Cipher decCipher = Cipher.getInstance("AES/GCM/NOPADDING");
        decCipher.init(Cipher.DECRYPT_MODE, key, ap);
        byte[] dec = decCipher.doFinal(enc);

        if (Arrays.compare(data, dec) != 0) {
            throw new Exception("Original data != decrypted data");
        }
    }
}
```

HMAC-SHA256 Example

The following is a sample program that demonstrates how to generate a secret-key object for HMAC-SHA256, and initialize a HMAC-SHA256 object with it.

Example 2-14 Generate a Secret-key Object for HMAC-SHA256

```
import java.security.*;
import javax.crypto.*;

    /**
     * This program demonstrates how to generate a secret-key object for
     * HMACSHA256, and initialize an HMACSHA256 object with it.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 */

public class initMac {

    public static void main(String[] args) throws Exception {

        // Generate secret key for HmacSHA256
        KeyGenerator kg = KeyGenerator.getInstance("HmacSHA256");
        SecretKey sk = kg.generateKey();

        // Get instance of Mac object implementing HmacSHA256, and
        // initialize it with the secret key, sk
        Mac mac = Mac.getInstance("HmacSHA256");
        mac.init(sk);
        byte[] result = mac.doFinal("Hi There".getBytes());
    }
}
```

Securely Exchanging Messages with HPKE and the HPKEParameterSpec Class

Hybrid Public Key Encryption (HPKE) is a public key encryption scheme for encrypting arbitrary-sized plaintexts with a recipient's public key. It combines a key encapsulation mechanism (KEM), a key derivation function (KDF), and an authenticated encryption with additional data (AEAD) cipher. Consequently, specifying the parameters for a HPKE cipher is more involved than other ciphers.

The `HPKEParameterSpec` class is an immutable class that specifies the set of parameters used with a `Cipher` for the Hybrid Public Key Encryption (HPKE) algorithm. See RFC 9180: Hybrid Public Key Encryption.

The following example uses HPKE with an X25519 key pair to securely exchange messages between a sender and a recipient:

```
// Recipient key pair generation
KeyPairGenerator g = KeyPairGenerator.getInstance("X25519");
KeyPair kp = g.generateKeyPair();

// The HPKE sender cipher is initialized with the recipient's public
// key and an HPKEParameterSpec using specified algorithm identifiers
// and application-supplied info.
Cipher senderCipher = Cipher.getInstance("HPKE");
HPKEParameterSpec ps = HPKEParameterSpec.of(
    HPKEParameterSpec.KEM_DHKEM_X25519_HKDF_SHA256,
    HPKEParameterSpec.KDF_HKDF_SHA256,
    HPKEParameterSpec.AEAD_AES_128_GCM)
   .withInfo(HexFormat.of().parseHex("010203040506"));
senderCipher.init(Cipher.ENCRYPT_MODE, kp.getPublic(), ps);

// Retrieve the key encapsulation message (from the KEM step) from
// the sender.
byte[] kemEncap = senderCipher.getIV();

// The HPKE recipient cipher is initialized with its own private key,
// an HPKEParameterSpec using the same algorithm identifiers as used by
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
// the sender, and the key encapsulation message from the sender.
Cipher recipientCipher = Cipher.getInstance("HPKE");
HPKEParameterSpec pr = HPKEParameterSpec.of(
    HPKEParameterSpec.KEM_DHKEM_X25519_HKDF_SHA256,
    HPKEParameterSpec.KDF_HKDF_SHA256,
    HPKEParameterSpec.AEAD_AES_128_GCM)
    .withInfo(HexFormat.of().parseHex("010203040506"))
    .withEncapsulation(kemEncap);
recipientCipher.init(Cipher.DECRYPT_MODE, kp.getPrivate(), pr);

// Encryption and decryption
byte[] msg = "Hello World".getBytes(StandardCharsets.UTF_8);
byte[] ct = senderCipher.doFinal(msg);
byte[] pt = recipientCipher.doFinal(ct);

assert Arrays.equals(msg, pt);
```

The standard name algorithm for the cipher is HPKE. See `Cipher` Algorithms in *Java Security Standard Algorithm Names*. Unlike most other ciphers, HPKE is not expressed as a transformation string of the form *algorithm*/*mode*/*padding*. Therefore, the argument to `Cipher.getInstance` must be the single algorithm name `HPKE`, for example:

```
Cipher senderCipher = Cipher.getInstance("HPKE");
// ...
Cipher recipientCipher = Cipher.getInstance("HPKE");
```

In HPKE, the sender's `Cipher` is always initialized with the recipient's public key in encrypt mode (see `Cipher.ENCRYPT_MODE`, while the recipient's `Cipher` object is initialized with its own private key in decrypt mode (see `Cipher.DECRYPT_MODE`), for example:

```
senderCipher.init(Cipher.ENCRYPT_MODE, kp.getPublic(), ps);
// ...
recipientCipher.init(Cipher.DECRYPT_MODE, kp.getPrivate(), pr)
```

An `HPKEParameterSpec` object must be provided when a HPKE cipher is initialized with the `Cipher.init(int opmode, Key, AlgorithmParameterSpec)` method.

The `HPKEParameterSpec.of(int kem_id, int kdf_id, int aead_id)` static method returns an `HPKEParameterSpec` with the specified KEM, KDF, and AEAD algorithm identifiers, for example:

```
HPKEParameterSpec ps = HPKEParameterSpec.of(
    HPKEParameterSpec.KEM_DHKEM_X25519_HKDF_SHA256,
    HPKEParameterSpec.KDF_HKDF_SHA256,
    HPKEParameterSpec.AEAD_AES_128_GCM)
   .withInfo(HexFormat.of().parseHex("010203040506"));
```

These algorithm identifiers are defined in 7. Algorithm Identifiers in RFC 9180, which is maintained bu IANA; see their page Hybrid Public Key Encryption (HPKE).

Once an `HPKEParameterSpec` object is created, you can use additional methods to generate new `HPKEParameterSpec` objects with different features:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- You can specify application-supplied information, the `info` parameter, which can be used to influence the generation of keys, by calling the `HPKEParameterSpec.withInfo(byte[] info)` method.

  In this example, the sender's `HPKEParameterSpec` `info` parameter is a `byte` array containing the values 1 through 6. In a production environment, the `info` parameter might include identity information instead.
- To authenticate using a preshared key with PSK mode, you must provide the preshared key, the `psk` parameter, and its identifier, the `psk_id` parameter, with the `HPKEParameterSpec.withPsk(SecretKey psk, byte[] psk_id)` method.
- To authenticate using an asymmetric key, you must provide it with the `HPKEParameterSpec.withAuthKey(AsymmetricKey)` method. Precisely, the sender must call this method with its own private key, and the recipient must call it with the sender's public key.
- To authenticate using both a PSK and an asymmetric key, you must provide them with `withAuthKey(AsymmetricKey)` and `withPsk(SecretKey, byte[])` as described previously.
- In HPKE, a shared secret is negotiated during the KEM step and a key encapsulation message must be transmitted from the sender to the recipient so that the recipient can recover the shared secret. On the sender side, after the cipher is initialized, retrieve the key encapsulation message with the `Cipher.getIV()` method, for example:

  ```
  byte[] kemEncap = senderCipher.getIV();
  ```

  On the recipient side, this message must be supplied as part of an `HPKEParameterSpec` object obtained from the `HPKEParameterSpec.withEncapsulation(byte[])` method, for example:

  ```
  HPKEParameterSpec pr = HPKEParameterSpec.of(
      HPKEParameterSpec.KEM_DHKEM_X25519_HKDF_SHA256,
      HPKEParameterSpec.KDF_HKDF_SHA256,
      HPKEParameterSpec.AEAD_AES_128_GCM)
      .withInfo(HexFormat.of().parseHex("010203040506"))
      .withEncapsulation(kemEncap);
  ```

For successful interoperability, both the sender and the recipient must have identical algorithm identifiers and supply identical `info`, `psk`, and `psk_id` parameters or matching authentication keys if provided.

Copyright © 1993, 2026, Oracle and/or its affiliates.

3

How to Implement a Provider in the Java Cryptography Architecture

This document describes what you need to do in order to integrate your provider into Java SE so that algorithms and other services can be found when Java Security API clients request them.

Who Should Read This Document

Programmers who only need to use the Java Security APIs (see Core Classes and Interfaces in Java Cryptography Architecture (JCA) Reference Guide) to access existing cryptography algorithms and other services do *not* need to read this document.

This document is intended for experienced programmers wishing to create their own provider packages supplying cryptographic service implementations. It documents what you need to do in order to integrate your provider into Java so that your algorithms and other services can be found when Java Security API clients request them.

Notes on Terminology

Throughout this document, the terms *JCA* by itself refers to the JCA framework. Whenever this document notes a specific JCA provider, it will be referred to explicitly by the provider name.

- Prior to JDK 1.4, the JCE was an unbundled product, and as such, the JCA and JCE were regularly referred to as separate, distinct components. As JCE is now bundled in JDK, the distinction is becoming less apparent. Since the JCE uses the same architecture as the JCA, the JCE should be more properly thought of as a subset of the JCA.
- The JCA within the JDK includes two software components:
  - the framework that defines and supports cryptographic services for which providers supply implementations. This framework includes packages such as `java.security`, `javax.crypto`, `javax.crypto.spec`, and `javax.crypto.interfaces`.
  - the actual providers such as *Sun*, *SunRsaSign*, *SunJCE*, which contain the actual cryptographic implementations.
- The JCE consists of the `javax.crypto.*` packages and the SunJCE provider.

Introduction to Implementing Providers

The Java platform defines a set of APIs spanning major security areas, including cryptography, public key infrastructure, authentication, secure communication, and access control. These APIs allow developers to easily integrate security into their application code. They were designed around the following principles:

- **Implementation independence**: Applications do not need to implement security themselves. Rather, they can request security services from the Java platform. Security services are implemented in providers, which are plugged into the Java platform via a

Copyright © 1993, 2026, Oracle and/or its affiliates.

standard interface. An application may rely on multiple independent providers for security functionality.

- **Implementation interoperability**: Providers are interoperable across applications. Specifically, an application is not bound to a specific provider, and a provider is not bound to a specific application.
- **Algorithm extensibility**: The Java platform includes a number of built-in providers that implement a basic set of security services that are widely used today. However, some applications may rely on emerging standards not yet implemented, or on proprietary services. The Java platform supports the installation of custom providers that implement such services.

A Cryptographic Service Provider (provider) refers to a package (or a set of packages) that supply a concrete implementation of a subset of the cryptography aspects of the JDK Security API.

The `java.security.Provider` class encapsulates the notion of a security provider in the Java platform. It specifies the provider's name and lists the security services it implements. Multiple providers may be configured at the same time, and are listed in order of preference. When a security service is requested, the highest priority provider that implements that service is selected. See Security Providers, which illustrates how a provider selects a requested security service.

Engine Classes and Corresponding Service Provider Interface Classes

An engine class defines a cryptographic service in an abstract fashion (without a concrete implementation). A cryptographic service is always associated with a particular algorithm or type.

A *cryptographic service* either provides cryptographic operations (like those for digital signatures or message digests, ciphers or key agreement protocols); generates or supplies the cryptographic material (keys or parameters) required for cryptographic operations; or generates data objects (keystores or certificates) that encapsulate cryptographic keys (which can be used in a cryptographic operation) in a secure fashion.

The Java Cryptography Architecture encompasses the classes comprising the Security package that relate to cryptography, including *engine classes*. Users of the API request and utilize instances of the engine classes to carry out corresponding operations. The JDK defines the following engine classes:

- `AlgorithmParameterGenerator` - used to generate a set of parameters suitable for a specified algorithm.
- `AlgorithmParameters` - used to manage the parameters for a particular algorithm, including parameter encoding and decoding.
- `CertificateFactory` - used to create public key certificates and Certificate Revocation Lists (CRLs).
- `CertPathBuilder` - used to create public key certificates and Certificate Revocation Lists (CRLs).
- `CertPathValidator` - used to validate certificate chains.
- `CertStore` - used to retrieve Certificates and CRLs from a repository.
- `Cipher` - used to encrypt or decrypt some specified data. It provides access to the functionality of an encryption algorithm (such as AES).

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `ExemptionMechanism` - used to provide the functionality of an exemption mechanism such as *key recovery*, *key weakening*, *key escrow*, or any other (custom) exemption mechanism. Applications that use an exemption mechanism may be granted stronger encryption capabilities than those which don't. However, please note that cryptographic restrictions are no longer required for most countries, and thus exemption mechanisms may only be useful in those few countries whose governments mandate restrictions.
- `KDF` - used to provide the functionality of a Key Derivation Function (KDF), which uses cryptographic inputs to create new cryptographically strong key material.
- `KEM` - used to provide the functionality of a Key Encapsulation Mechanism (KEM). A KEM can be used to secure symmetric keys using asymmetric or public key cryptography between two parties. The sender calls the encapsulate method to generate a secret key and a key encapsulation message, and the receiver calls the decapsulate method to recover the same secret key from the key encapsulation message.
- `KeyAgreement` - used to execute a key agreement (key exchange) protocol between two or more parties. It provides access to the functionality of a key agreement protocol (such as Diffie-Hellman)
- `KeyFactory` - used to convert opaque cryptographic keys of type `Key` into *key specifications* (transparent representations of the underlying key material), and vice versa. A DSA `KeyFactory` class supplies a DSA private or public key (from its encoding or transparent specification) in a format usable by the `initSign` or `initVerify` methods, respectively, of a DSA `Signature` object.
- `KeyGenerator` - used to generate a secret (symmetric) key suitable for a specified algorithm.
- `KeyPairGenerator` - used to generate a pair of public and private keys suitable for a specified algorithm.
- `KeyStore` - used to create and manage a *keystore*. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.
- `Mac`: used to compute the *message authentication code* of some specified data.
- `MessageDigest` - used to calculate the message digest (hash) of specified data.
- `SecretKeyFactory` - used to convert opaque cryptographic keys of type `SecretKey` into *key specifications* (transparent representations of the underlying key material), and vice versa.
- `SecureRandom` - used to generate random or pseudo-random numbers.
- `Signature` - used to sign data and verify digital signatures. It provides access to the functionality of a digital signature algorithm.

> **Note**
>
> A *generator* creates objects with brand-new contents, whereas a *factory* creates objects from existing material (for example, an encoding).

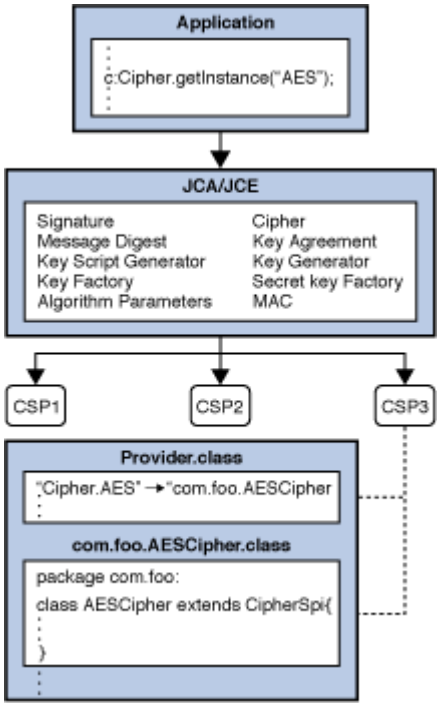
An *engine* class provides the interface to the functionality of a specific type of cryptographic service (independent of a particular cryptographic algorithm). It defines *Application Programming Interface (API)* methods that allow applications to access the specific type of cryptographic service it provides. The actual implementations (from one or more providers) are those for specific algorithms. For example, the Signature engine class provides access to the functionality of a digital signature algorithm. The actual implementation supplied in a

Copyright © 1993, 2026, Oracle and/or its affiliates.

`SignatureSpi` subclass (see next paragraph) would be that for a specific kind of signature algorithm, such as SHA256withDSA or SHA512withRSA.

The application interfaces supplied by an engine class are implemented in terms of a **Service Provider Interface (SPI)**. That is, for each engine class, there is a corresponding abstract SPI class, which defines the Service Provider Interface methods that cryptographic service providers must implement.

Figure 3-1 Engine Classes



An instance of an engine class, the "API object", encapsulates (as a private field) an instance of the corresponding SPI class, the "SPI object". All API methods of an API object are declared "final", and their implementations invoke the corresponding SPI methods of the encapsulated SPI object. An instance of an engine class (and of its corresponding SPI class) is created by a call to the `getInstance` factory method of the engine class.

The name of each SPI class is the same as that of the corresponding engine class, followed by "Spi". For example, the SPI class corresponding to the Signature engine class is the `SignatureSpi` class.

Each SPI class is abstract. To supply the implementation of a particular type of service and for a specific algorithm, a provider must subclass the corresponding SPI class and provide implementations for all the abstract methods.

Another example of an engine class is the `MessageDigest` class, which provides access to a message digest algorithm. Its implementations, in `MessageDigestSpi` subclasses, may be those of various message digest algorithms such as SHA-256 or SHA-384.

As a final example, the `KeyFactory` engine class supports the conversion from opaque keys to transparent key specifications, and vice versa. See Key Specification Interfaces and Classes Required by Key Factories. The actual implementation supplied in a `KeyFactorySpi` subclass would be that for a specific type of keys, such as DSA public and private keys.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Steps to Implement and Integrate a Provider

Follow these steps to implement a provider and integrate it into the JCA framework:

- Step 1: Write your Service Implementation Code
- Step 2: Give your Provider a Name
- Step 3: Write Your Master Class, a Subclass of Provider
- Step 4: Create a Module Declaration for Your Provider
- Step 5: Compile Your Code
- Step 6: Place Your Provider in a JAR File
- Step 7: Sign Your JAR File, If Necessary
- Step 8: Prepare for Testing
- Step 9: Write and Compile Your Test Programs
- Step 10: Run Your Test Programs
- Step 11: Apply for U.S. Government Export Approval If Required
- Step 12: Document Your Provider and Its Supported Services
- Step 13: Make Your Class Files and Documentation Available to Clients

Step 1: Write your Service Implementation Code

The first thing you need to do is to write the code that provides algorithm-specific implementations of the cryptographic services you want to support. Your provider may supply implementations of cryptographic services already available in one or more of the security components of the JDK.

For cryptographic services not defined in JCA (for example, signatures and message digests), see Engine Classes and Algorithms.

For each cryptographic service you wish to implement, create a subclass of the appropriate SPI class. JCA defines the following engine classes:

- `AlgorithmParameterGeneratorSpi`
- `AlgorithmParametersSpi`
- `CertificateFactorySpi`
- `CipherSpi`
- `ExemptionMechanismSpi`
- `KDFSpi`
- `KEMSpi`
- `KeyAgreementSpi`
- `KeyFactorySpi`
- `KeyGeneratorSpi`
- `KeyPairGeneratorSpi`
- `KeyStoreSpi`

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `MacSpi`
- `MessageDigestSpi`
- `SecretKeyFactorySpi`
- `SecureRandomSpi`
- `SignatureSpi`

To know more about the JCA and other cryptographic classes, see Engine Classes and Corresponding Service Provider Interface Classes.

In the subclass, you need to:

1. Supply implementations for the abstract methods, whose names usually begin with `engine`. See Further Implementation Details and Requirements.
2. Depending on how you write your provider and register its algorithms (using either `String` objects or the `Provider.Service` class), the provider either:
   - Ensure that there is a public constructor without any arguments. Here's why: When one of your services is requested, Java Security looks up the subclass implementing that service, as specified by a property in your "master class" (see Step 3: Write Your Master Class, a Subclass of Provider). Java Security then creates the `Class` object associated with your subclass, and creates an instance of your subclass by calling the `newInstance` method on that `Class` object. `newInstance` requires your subclass to have a public constructor without any parameters. (A default constructor without arguments will automatically be generated if your subclass doesn't have any constructors. But if your subclass defines any constructors, you must explicitly define a public constructor without arguments.)
   - Override the `newInstance()` method in the registered `Provider.Service`. This is the preferred mechanism in JDK 9 and later.

Step 1.1: Consider Additional JCA Provider Requirements and Recommendations for Encryption Implementations

When instantiating a provider's implementation (class) of a `Cipher`, `KDF`, `KEM`, `KeyAgreement`, `KeyGenerator`, `MAC`, or `SecretKeyFactory`, the framework will determine the provider's codebase (JAR file) and verify its signature. In this way, JCA authenticates the provider and ensures that only providers signed by a trusted entity can be plugged into the JCA. Thus, one requirement for encryption providers is that they must be signed, as described in later steps.

In order for provider classes to become unusable if instantiated by an application directly, bypassing JCA, providers should implement the following:

- All SPI implementation classes in a provider package should be declared `final` (so that they cannot be subclassed), and their (SPI) implementation methods should be declared `protected`.
- All crypto-related helper classes in a provider package should have package-private scope, so that they cannot be accessed from outside the provider package.

For providers that may be exported outside the U.S., `CipherSpi` implementations must include an implementation of the `engineGetKeySize` method which, given a `Key`, returns the key size. If there are restrictions on available cryptographic strength specified in jurisdiction policy files, each `Cipher` initialization method calls `engineGetKeySize` and then compares the result with the maximum allowable key size for the particular location and circumstances of the application being run. If the key size is too large, the initialization method throws an exception.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Additional optional features that providers may implement are:

- Optional: The `engineWrap` and `engineUnwrap` methods of `CipherSpi`. Wrapping a key enables secure transfer of the key from one place to another. See the `wrap` method for more information about wrapping and unwrapping keys.
- Optional: One or more *exemption mechanisms*. An exemption mechanism is something such as key recovery, key escrow, or key weakening which, if implemented and enforced, may enable reduced cryptographic restrictions for an application that uses it. To know more about the requirements for apps that utilize exemption mechanisms, see How to Make Applications Exempt from Cryptographic Restrictions.

Step 2: Give your Provider a Name

Decide on a unique name for your provider. This is the name to be used by client applications to refer to your provider, and it must not conflict with any other provider names.

Step 3: Write Your Master Class, a Subclass of Provider

Create a subclass of the `java.security.Provider` class. This is essentially a lookup table that advertises the algorithms that your provider implements.

You can use the following coding styles to subclass the `Provider` class:

- Create a provider that registers its services with `String` objects to store algorithm names and their associated implementation class name. These are stored in the `Hashtable<Object,Object>` superclass of `java.security.Provider`.
- Create a provider that uses the `Provider.Service` class, which uses a different method to store algorithm names and create new objects. The `Provider.Service` class enables you customize how the JCA framework requests services from your provider, such as how the framework creates new instances of your provider's services. This coding style is recommended, especially when using modules.

A provider can use either style, or even use both styles at the same time. Regardless of which style you choose, your subclass should be `final`.

Step 3.1: Create a Provider That Uses String Objects to Register Its Services

The following is an example of a provider that uses `String` objects to store implemented algorithm names:

```
package p;
public final class MyProvider extends Provider {
    public MyProvider() {
        super("MyProvider", "1.0",
            "Some info about my provider and which algorithms it supports");
        // com.my.crypto.provider.MyCipher extends CipherSPI
        put("Cipher.MyCipher", "com.my.crypto.provider.MyCipher");
    }
}
```

To create a provider with this coding style, do the following:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Call `super`, specifying the provider name (see Step 2: Give your Provider a Name) version number, and a string of information about the provider and algorithms it supports.

  ```
  super("MyProvider", "1.0",
        "Some info about my provider and which algorithms it supports");
  ```

- Set the values of various properties that are required for the Java Security API to look up the cryptographic services implemented by the provider.

  For each service implemented by the provider, there must be a property whose name is the type of service followed by a period and the name of the algorithm to which the service applies. The property value must specify the fully qualified name of the class implementing the service.

  For example, this following statement sets a property named `Cipher.MyCypher` whose value is `com.my.crypto.provider.MyCipher`, a class that extends `CipherSPI`:

  ```
  put("Cipher.MyCipher", "com.my.crypto.provider.MyCipher");
  ```

  The following list shows the various types of JCA services, where the actual algorithm name is substituted for *`algName`*:

  - `AlgorithmParameterGenerator.`*`algName`*
  - `AlgorithmParameters.`*`algName`*
  - `CertificateFactory.`*`algName`*
  - `Cipher.`*`algName`*

    > **Note**
    >
    > *`algName`* may actually represent a transformation, and may be composed of an algorithm name, a particular mode, and a padding scheme. See Java Security Standard Algorithm Names.

  - `ExemptionMechanism.`*`algName`*:

    > **Note**
    >
    > *`algName`* refers to the name of the exemption mechanism, which can be one of the following: `KeyRecovery`, `KeyEscrow`, or `KeyWeakening`. Case does *not* matter.

  - `KDF.`*`algName`*
  - `KEM.`*`algName`*
  - `KeyAgreement.`*`algName`*
  - `KeyFactory.`*`algName`*
  - `KeyGenerator.`*`algName`*
  - `KeyPairGenerator.`*`algName`*
  - `KeyStore.`*`algName`*
  - `Mac.`*`algName`*

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `MessageDigest.algName`
- `SecretKeyFactory.algName`
- `SecureRandom.algName`
- `Signature.algName`

In all of these except `ExemptionMechanism` and `Cipher`, `algName` is the "standard" name of the algorithm, certificate type, or keystore type. See Java Security Standard Algorithm Names for the standard names that should be used.

The value of each property must be the fully qualified name of the class implementing the specified algorithm, certificate type, or keystore type. That is, it must be the package name followed by the class name, where the two are separated by a period.

As an example, the default provider named *SUN* implements the Digital Signature Algorithm (whose standard name is `SHA256withDSA`) in a class named `DSA` in the `sun.security.provider` package. Its subclass of `Provider` (which is the Sun class in the `sun.security.provider package`) sets the `Signature.SHA256withDSA` property to have the value `sun.security.provider.DSA` via the following:

```
put("Signature.SHA256withDSA", "sun.security.provider.DSA");
```

The following list shows more properties that can be defined for the various types of services, where the actual algorithm name is substituted for *algName*, certificate type for *certType,* keystore type for *storeType*, and attribute name for *attrName*:

- `AlgorithmParameterGenerator.algName attrName`
- `AlgorithmParameters.algName attrName`
- `CertificateFactory.certType attrName`
- `Cipher.algName attrName`
- `ExemptionMechanism.algName attrName`
- `KDF.algName attrName`
- `KEM.algName attrName`
- `KeyAgreement.algName attrName`
- `KeyFactory.algName attrName`
- `KeyGenerator.algName attrName`
- `KeyPairGenerator.algName attrName`
- `KeyStore.storeType attrName`
- `Mac.algName attrName`
- `MessageDigest.algName attrName`
- `SecretKeyFactory.algName attrName`
- `SecureRandom.algName attrName`
- `Signature.algName attrName`

In each of these, *attrName* is the "standard" name of the algorithm, certificate type, keystore type, or attribute. (See Java Security Standard Algorithm Names for the standard names that should be used.)

Copyright © 1993, 2026, Oracle and/or its affiliates.

For a property in this format, the value of the property must be the value for the corresponding attribute. (See Java Security Standard Algorithm Names for the definition of each standard attribute.)

For further master class property setting examples, see the JDK source code for the `sun.security.provider.Sun` and `com.sun.crypto.provider.SunJCE` classes. They show how the Sun and SunJCE providers set properties.

As an example, the default provider named SUN implements the `SHA256withDSA` Digital Signature Algorithm in software. The class `sun.security.provider.Sun` calls the method `SunEntries.putEntries`, which sets the properties for the SUN provider, including setting the property `Signature.SHA256withDSA ImplementedIn` to have the value `Software`:

```
put("Signature.SHA256withDSA ImplementedIn", "Software");
```

> **Note**
>
> For examples of this coding style, see the source code for `sun.security.provider.Sun` and `sun.security.provider.SunEntries` classes.

Step 3.2: Create a Provider That Uses Provider.Service

The following is an example of a provider that uses a `Provider.Service` class:

```
package p;

public final class MyProvider extends Provider {

    public MyProvider() {
        super("MyProvider", "1.0",
            "Some info about my provider and which algorithms it supports");
        putService(new ProviderService(this, "Cipher", "MyCipher",
"p.MyCipher"));
    }

    private static final class ProviderService extends Provider.Service {
        ProviderService(Provider p, String type, String algo, String cn) {
            super(p, type, algo, cn, null, null);
        }

        @Override
        public Object newInstance(Object ctrParamObj)
            throws NoSuchAlgorithmException {
            String type = getType();
            String algo = getAlgorithm();
            try {
                if (type.equals("Cipher")) {
                    if (algo.equals("MyCipher")) {
                        return new MyCipher();
                    }
                }
            } catch (Exception ex) {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                throw new NoSuchAlgorithmException(
                    "Error constructing " + type + " for "
                    + algo + " using MyProvider", ex);
            }
            throw new ProviderException("No impl for " + algo + " " + type);
        }
    }
}
```

To create a provider with this coding style, do the following:

- For each algorithm your provider supports, call `putService` with an instance of `Provider.Service`; the arguments of the `Provider.Service` constructor represent a supported algorithm.

  The following statement adds a service named `MyCipher` of type `Cipher`; the name of the class implementing this service is `p.MyCipher`. The argument of `putService` is a subclass of `Provider.Service`:

  ```
          putService(new ProviderService(this, "Cipher", "MyCipher",
  "p.MyCipher"));
  ```

  This example uses a subclass of `Provider.Service` named `ProviderService` (rather than `Provider.Service` itself) as it customizes how the JCA framework instantiates services. If you don't need to customize the behavior of `Provider.Service`, then you can call the `Provider.Service` constructor directly:

  ```
  public final class MyProvider extends Provider {
      public MyProvider() {
          super("MyProvider", "1.0",
              "Some info about my provider and which algorithms it
  supports");
          putService(new Provider.Service(
              this, "Cipher", "MyCipher", "p.MyCipher", null, null));
      }
  }
  ```

  Note that this example is essentially the same as the example described in Step 3.1: Create a Provider That Uses String Objects to Register Its Services.

- Override any method in `Provider.Service`, such as `newInstance`, to customize how the JCA framework handles the services in your provider.

  The example at the beginning of this section overrides the method `Provider.Service.newInstance`. The method returns an instance of `MyCipher` only if the requested service is `MyCipher`. If not, it throws a `NoSuchAlgorithmException` and a `ProviderException`.

  For more information about other methods you can override, see The Provider.Service Class.

  > **Note**
  >
  > For examples of this coding style, see the JDK source code contained in the `sun.security.mscapi` package.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Step 3.3: Specify Additional Information for Cipher Implementations

As mentioned previously, in the case of a `Cipher` property, *algName* may actually represent a *transformation*. A *transformation* is a string that describes the operation (or set of operations) to be performed by a `Cipher` object on some given input. A transformation always includes the name of a cryptographic algorithm (e.g., *AES*), and may be followed by a mode and a padding scheme.

A transformation is of the form:

- *algorithm/mode/padding*, or
- *algorithm*

(In the latter case, provider-specific default values for the mode and padding scheme are used). For example, the following is a valid transformation:

```
Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

When requesting a block cipher in stream cipher mode (for example; `AES` in `CFB` or `OFB` mode), a client may optionally specify the number of bits to be processed at a time, by appending this number to the mode name as shown in the following sample transformations:

```
Cipher c1 = Cipher.getInstance("AES/CFB8/NoPadding");
Cipher c2 = Cipher.getInstance("AES/OFB32/PKCS5Padding");
```

If a number does not follow a stream cipher mode, a provider-specific default is used. (For example, the *SunJCE* provider uses a default of 128 bits.)

A provider may supply a separate class for each combination of *algorithm/mode/padding*. Alternatively, a provider may decide to provide more generic classes representing sub-transformations corresponding to *algorithm* or *algorithm/mode* or *algorithm//padding* (note the double slashes); in this case the requested mode and/or padding are set automatically by the `getInstance` methods of `Cipher`, which invoke the `engineSetMode` and `engineSetPadding` methods of the provider's subclass of `CipherSpi`.

That is, a `Cipher` property in a provider master class may have one of the formats shown in the following table:

Table 3-1 Cipher Property Format

`Cipher` Property Format	Description
`Cipher.`*algName*	A provider's subclass of `CipherSpi` implements *algName* with pluggable mode and padding
`Cipher.`*algName/mode*	A provider's subclass of `CipherSpi` implements *algName* in the specified *mode*, with pluggable padding
`Cipher.`*algName//padding*	A provider's subclass of `CipherSpi` implements *algName* with the specified *padding*, with pluggable mode
`Cipher.`*algName/mode/padding*	A provider's subclass of `CipherSpi` implements *algName* with the specified *mode* and *padding*

(See Java Security Standard Algorithm Names for the standard algorithm names, modes, and padding schemes that should be used.)

Copyright © 1993, 2026, Oracle and/or its affiliates.

For example, a provider may supply a subclass of `CipherSpi` that implements *AES/ECB/PKCS5Padding*, one that implements *AES/CBC/PKCS5Padding*, one that implements *AES/CFB/PKCS5Padding*, and yet another one that implements *AES/OFB/PKCS5Padding*. That provider would have the following `Cipher` properties in its master class:

- `Cipher.`*`AES/ECB/PKCS5Padding`*
- `Cipher.`*`AES/CBC/PKCS5Padding`*
- `Cipher.`*`AES/CFB/PKCS5Padding`*
- `Cipher.`*`AES/OFB/PKCS5Padding`*

Another provider may implement a class for each of these modes (for example, one class for *ECB*, one for *CBC*, one for *CFB*, and one for *OFB*), one class for *PKCS5Padding*, and a generic *AES* class that subclasses from `CipherSpi`. That provider would have the following `Cipher` properties in its master class:

- `Cipher.`*`AES`*
- `Cipher.`*`AES`* `SupportedModes`
  - Example: `"ECB|CBC|CFB|OFB"`
- `Cipher.`*`AES`* `SupportedPaddings`
  - Example: `"NOPADDING|PKCS5Padding"`

The `getInstance` factory method of the `Cipher` engine class follows these rules in order to instantiate a provider's implementation of `CipherSpi` for a transformation of the form "*algorithm*":

1. Check if the provider has registered a subclass of `CipherSpi` for the specified "*algorithm*".
   - If the answer is YES, instantiate this class, for whose mode and padding scheme default values (as supplied by the provider) are used.
   - If the answer is NO, throw a `NoSuchAlgorithmException` exception.
2. The `getInstance` factory method of the `Cipher` engine class follows these rules in order to instantiate a provider's implementation of `CipherSpi` for a transformation of the form "*algorithm/mode/padding*":
   a. Check if the provider has registered a subclass of `CipherSpi` for the specified "*algorithm/mode/padding*" transformation.
      - If the answer is YES, instantiate it.
      - If the answer is NO, go to the next step.
   b. Check if the provider has registered a subclass of `CipherSpi` for the sub-transformation "*algorithm/mode*".
      - If the answer is YES, instantiate it, and call `engineSetPadding(`*`padding`*`)` on the new instance.
      - If the answer is NO, go to the next step.
   c. Check if the provider has registered a subclass of `CipherSpi` for the sub-transformation "*algorithm//padding*" (note the double slashes)
      - If the answer is YES, instantiate it, and call `engineSetMode(`*`mode`*`)` on the new instance.
      - If the answer is NO, go to the next step.
   d. Check if the provider has registered a subclass of `CipherSpi` for the sub-transformation "*algorithm*".

Copyright © 1993, 2026, Oracle and/or its affiliates.

- If the answer is YES, instantiate it, and call `engineSetMode(`*`mode`*`)` and `engineSetPadding(`*`padding`*`)` on the new instance.
- If the answer is NO, throw a `NoSuchAlgorithmException` exception.

Step 4: Create a Module Declaration for Your Provider

This step is optional but recommended; it enables you to package your provider in a named module. A modular JDK can then locate your provider in the module path as opposed to the class path. The module system can more thoroughly check for dependencies in modules in the module path. Note that you can use named modules in a non-modular JDK; the module declaration will be ignored. Also, you can still package your providers in unnamed or automatic modules.

Create a module declaration for your provider and save it in a file named `module-info.java`. This module declaration includes the following:

- The name of your module.
- Any module upon which your provider depends.
- A `provides` directive if your module provides a service implementation.

The following example module declaration defines a module named `com.foo.MyProvider`. `p.MyProvider` is the fully qualified class name of a service implementation. Suppose that, in this example, `p.MyProvider` uses API in the package `javax.security.auth.kerberos`, which is in the module `java.security.jgss`. Thus, the directive `requires java.security.jgss` appears in the module declaration.

```
module com.foo.MyProvider {
    provides java.security.Provider with p.MyProvider;
    requires java.security.jgss;
    }
```

You can package a provider in three different kinds of modules:

- Named or explicit module: A module that appears on the module path and contains module configuration information in the `module-info.class` file.

  The JCA framework can use the `ServiceLoader` class (which simplifies provider configuration) to search for providers in explicit modules without any additional changes to the module. See Step 8.1: Configure the Provider and Step 10: Run Your Test Programs.
- Automatic module: A module that appears on the module path, but does not contain module configuration information in a `module-info.class` file (essentially a "regular" JAR file).
- Unnamed module: A module that appears on the class path. It may or may not have a `module-info.class` file; this file is ignored.

It is recommended that you package your providers in named modules as they provide better performance, stronger encapsulation, simpler configuration and greater flexibility.

You have a lot of flexibility when it comes to packaging and configuring your providers. However, this impacts how you start applications that use them. For example, you might have to specify additional `--add-exports` or `--add-modules` options. Named modules, in general, require fewer of these additional options. In addition named modules offer more flexibility. You can use them with non-modular JDKs or even as unnamed modules by specifying them in a modular JDK's class path. For more information about modules, see The State of the Module System and JEP 261: Module System.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Step 5: Compile Your Code

After you have created your implementation code (Step 1: Write your Service Implementation Code), given your provider a name (Step 2: Give your Provider a Name), created the master class (Step 3: Write Your Master Class, a Subclass of Provider), and created a module declaration (Step 4: Create a Module Declaration for Your Provider), use the Java compiler to compile your files.

Step 6: Place Your Provider in a JAR File

Add the File java.security.Provider to Use the ServiceLoader Class to Search for Providers

If your provider is packaged in an automatic or unnamed module (you did not create a module declaration as described in Step 4: Create a Module Declaration for Your Provider) and you want the use the `java.util.ServiceLoader` to search for your providers, then add the file `META-INF/services/java.security.Provider` to the JAR file and ensure that the file contains the fully qualified class name of your provider implementation.

The security provider loading mechanism uses the `ServiceLoader` class to search for providers before consulting the class path.

For example, if the fully qualified class name of your provider is `p.Provider` and all the compiled code of your provider is in the directory `classes`, then create a file named `classes/META-INF/services/java.security.Provider` that contains the following line:

```
p.MyProvider
```

Run the jar Command to Create a JAR File

The following command creates a JAR file named `MyProvider.jar`. All the compiled code for the module JAR file is in the directory `classes`. In addition, the module descriptor, `module-info.class`, is in the directory `classes`:

```
jar --create --file MyProvider.jar --module-version 1.0 -C classes
```

> **Note**
>
> The `module-info.class` file and the `--module-version` option are optional. However, the `module-info.class` file is required if you want to create a modular JAR file. (A modular JAR file is a regular JAR file that has a `module-info.class` file in its top-level directory.)

See `jar` in *Java Development Kit Tool Specifications*.

Step 7: Sign Your JAR File, If Necessary

If your provider is supplying encryption algorithms through the `Cipher`, `KDF`, `KEM`, `KeyAgreement`, `KeyGenerator`, `Mac`, or `SecretKeyFactory` classes, you must sign your

Copyright © 1993, 2026, Oracle and/or its affiliates.

JAR file so that the JCA can authenticate the code at run time; see Step 1.1: Consider Additional JCA Provider Requirements and Recommendations for Encryption Implementations. If you are not providing an implementation of this type, then you can skip this step.

Step 7.1: Get a Code-Signing Certificate

The next step is to request a code-signing certificate so that you can use it to sign your provider prior to testing. The certificate will be good for both testing and production. It will be valid for 5 years.

The following are the steps you should use to get a code-signing certificate. See `keytool` in the *Java Development Kit Tool Specifications*.

1. Use **keytool** to generate a 2048-bit RSA keypair:

```
keytool -genkeypair -alias <alias> \
        -keyalg RSA -keysize 2048 \
        -dname "cn=<Company Name>, \
        ou=Java Software Code Signing, \
        o=Oracle Corporation" \
        -keystore <keystore file name> \
        -storepass <keystore password>
```

This generates a 2048-bit RSA keypair (a public key and an associated private key) and stores it in an entry in the specified keystore. The public key is stored in a self-signed certificate. The keystore entry can subsequently be accessed using the specified alias.

> **Note**
>
> It's recommended that you create a keypair that uses RSA or DSA with 2048 or more bits.

The option values in angle brackets (< and >) represent the actual values that must be supplied. For example, *<alias>* must be replaced with whatever alias name you wish to be used to refer to the newly-generated keystore entry in the future, and *<keystore file name>* must be replaced with the name of the keystore to be used.

> **Tip**
>
> Do not surround actual values with angle brackets. For example, if you want your alias to be `myTestAlias`, specify the `-alias` option as follows:
>
> ```
> -alias myTestAlias
> ```

If you specify a keystore that doesn't yet exist, it will be created.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> If command lines you type are not allowed to be as long as the `keytool -genkeypair` command you want to execute (for example, if you are typing to a Microsoft Windows DOS prompt), you can create and execute a plain-text batch file containing the command. That is, create a new text file that contains nothing but the full `keytool -genkeypair` command. (Remember to type it all on one line.) Save the file with a .bat extension. Then in your DOS window, type the file name (with its path, if necessary). This will cause the command in the batch file to be executed.

2. Use **keytool** to generate a Certificate Signing Request (CSR):

```
keytool -certreq -alias <alias> \
    -file <csr file name> \
    -keystore <keystore file name> \
    -storepass <keystore password>
```

   Here, *<alias>* is the alias for the RSA keypair entry created in the previous step. This command generates a CSR, using the PKCS#10 format. It stores the CSR in the file whose name is specified in *<csr file name>*.

3. Request a JCE code signing certificate by sending your CSR, your contact information, and other required documentation to the JCA Code Signing Certification Authority. See JCA Code Signing Certification Authority for more information.

4. Once the JCE Code Signing Certification Authority receives your request, they will validate it and perform a background check. If this check passes, then they will create and sign a JCE code-signing certificate valid for 5 years. You will receive an email message containing two text certificates: the code-signing certificate and the JCE CA certificate, which authenticates the code-signing certificate's public key.

5. Import the certificates you received from the JCA Code Signing Certification Authority into your keystore with the keytool command.

   First import the CA's certificate as a "trusted certificate":

```
keytool -import -alias <alias for the CA cert> \
    -file <CA cert file name> \
    -keystore <keystore file name> \
    -storepass <keystore password>
```

   Then import the code-signing certificate:

```
keytool -import -alias <alias> \
    -file <code-signing cert file name> \
    -keystore <keystore file name> \
    -storepass <keystore password>
```

   <alias> is the same alias as that which you created in Step 1 where you generated a RSA keypair. This command replaces the self-signed certificate in the keystore entry specified by *<alias>* with the one signed by the JCA Code Signing Certification Authority.

Now that you have in your keystore a certificate from an entity trusted by JCA (the JCA Code Signing Certification Authority), you can place your provider code in a JAR file (Step 6: Place

Copyright © 1993, 2026, Oracle and/or its affiliates.

Your Provider in a JAR File) and then use that certificate to sign the JAR file (Step 7.2: Sign Your Provider).

Step 7.2: Sign Your Provider

Sign the JAR file created in Step 6: Place Your Provider in a JAR File with the code-signing certificate obtained in Step 7.1: Get a Code-Signing Certificate. See `jarsigner` in *Java Development Kit Tool Specifications*.

```
jarsigner -keystore <keystore file name> \
    -storepass <keystore password> \
    <JAR file name> <alias>
```

Here, `<alias>` is the alias into the keystore for the entry containing the code-signing certificate received from the JCA Code Signing Certification Authority (the same alias as that specified in the commands in Step 7.1: Get a Code-Signing Certificate).

You can test verification of the signature via the following:

```
jarsigner -verify <JAR file name>
```

The text `jar verified` will be displayed if the verification was successful.

> **Note**
>
> - You can also use the `jdk.security.jarsigner` API to sign JAR files.
> - If you include a signed JCE provider with your application and also want the JAR file signed for implementing other code-signing policies, you need to apply multiple signatures to the JCE provider JAR using the appropriate certificates/keys. The JCE signature is for acceptance of the provider JAR by the JCA framework, the other signature(s) can be used for making policy decisions. See `jarsigner` in *Java Development Kit Tool Specifications* for applying multiple signatures to a JAR file.
> - You cannot package signed providers in JMOD files.
> - Only providers that supply instances of `Cipher`, `KDF`, `KEM`, `KeyAgreement`, `KeyGenerator`, `Mac`, or `SecretKeyFactory` must be signed. If your provider only supplies other instances, such as `SecureRandom`, `MessageDigest`, `Signature`, and `KeyStore`, then the provider does not need to be signed.
> - You can link a provider in a custom runtime image with the `jlink` command as long as it doesn't have a `Cipher`, `KEM`, `KeyAgreement`, `KeyGenerator`, or `Mac` implementation.

Step 8: Prepare for Testing

The next steps describe how to install and configure your new provider so that it is available via the JCA.

Step 8.1: Configure the Provider

Register your provider so that the JCA framework can find your provider, either with the `ServiceLoader` class or in the class path or module path.

Copyright © 1993, 2026, Oracle and/or its affiliates.

1. Open the `java.security` file in an editor:
   - Linux or macOS: <*java-home*>`/conf/security/java.security`
   - Windows: <*java-home*>`\conf\security\java.security`
2. In the `java.security` file, find the section where standard providers such as SUN, SunRsaSign, and SunJCE are configured as static providers; it looks like the following:

   ```
   security.provider.1=SUN
   security.provider.2=SunRsaSign
   security.provider.3=SunEC
   security.provider.4=SunJSSE
   security.provider.5=SunJCE
   security.provider.6=SunJGSS
   security.provider.7=SunSASL
   security.provider.8=XMLDSig
   security.provider.9=SunPCSC
   security.provider.10=JdkLDAP
   security.provider.11=JdkSASL
   security.provider.12=SunMSCAPI
   security.provider.13=SunPKCS11
   ```

   Each line in this section has the following form:

   ```
   security.provider.n=provName|className
   ```

   This declares a provider, and specifies its preference order *n*. The preference order is the order in which providers are searched for requested algorithms when no specific provider is requested. The order is 1-based; 1 is the most preferred, followed by 2, and so on.

   provName is the provider's name and *className* is the fully qualified class name of the provider. You can use either of these two names.
3. Register your provider by adding to the `java.security` file a line with the form `security.provider.`*n*`=`*provName*`|`*className*.

   If you configured your provider so that the `ServiceLoader` class can search for it (because you packaged the provider in a named module as described in Step 4: Create a Module Declaration for Your Provider or added a `java.security.Provider` file as described in Add the File java.security.Provider to Use the ServiceLoader Class to Search for Providers), then specify just the provider's name.

   If you have not configured your provider so that `ServiceLoader` class can search for it, which means that the JCA framework will search for it in the class path or module path, then specify the fully qualified class name of your provider.

   For example, the highlighted line registers the provider `MyProvider` (whose fully qualified class name is `p.MyProvider` and has been configured so that the `ServiceLoader` class can search for it) as the 14th preferred provider:

   ```
   # ...
   security.provider.11=JdkSASL
   security.provider.12=SunMSCAPI
   security.provider.13=SunPKCS11
   security.provider.14=MyProvider
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

If you are not sure if the `ServiceLoader` mechanism will be used, or if you'll be deploying on a non-modular system, then you can also register the provider again, this time using the full class name:

```
security.provider.15=p.MyProvider
```

> **Note**
>
> Properties in the `java.security` file are typically parsed only once. If you have modified any property in this file, restart your applications to ensure that the changes are properly reflected.

Alternatively, you can register providers dynamically. To do so, a program (such as your test program, to be written in Step 9: Write and Compile Your Test Programs) call either the `addProvider` or `insertProviderAt` method in the `Security` class:

```
ServiceLoader<Provider> sl = ServiceLoader.load(java.security.Provider.class);
for (Provider p : sl) {
    System.out.println(p);
    if (p.getName().equals("MyProvider")) {
        Security.addProvider(p);
    }
}
```

Step 9: Write and Compile Your Test Programs

Write and compile one or more test programs that test your provider's incorporation into the Security API as well as the correctness of its algorithm(s). Create any supporting files needed, such as those for test data to be encrypted.

1. The first tests your program should perform are ones to ensure that your provider is found, and that its name, version number, and additional information is as expected.

   To do so, you could write code like the following, substituting your provider name for `MyPro`:

   ```
   Provider p = Security.getProvider("MyPro");
   System.out.println("MyPro provider name is " + p.getName());
   System.out.println("MyPro provider version # is " + p.getVersion());
   System.out.println("MyPro provider info is " + p.getInfo());
   ```

2. You should ensure that your services are found.

   For instance, if you implemented the AES encryption algorithm, you could check to ensure it's found when requested by using the following code (again substituting your provider name for "MyPro"):

   ```
   Cipher c = Cipher.getInstance("AES", "MyPro");
   System.out.println("My Cipher algorithm name is " + c.getAlgorithm());
   ```

3. Optional: If you don't specify a provider name in the call to `getInstance`, all registered providers will be searched, in preference order (see Step 8.1: Configure the Provider), until one implementing the algorithm is found.

Copyright © 1993, 2026, Oracle and/or its affiliates.

4. Optional: If your provider implements an exemption mechanism, you should write a test application that uses the exemption mechanism. Such an application also needs to be signed and have a "permission policy file" bundled with it.

   See How to Make Applications Exempt from Cryptographic Restrictions for complete information on creating and testing such an application.

Step 10: Run Your Test Programs

When you run your test applications, the required `java` command options will vary depending on factors such as whether you packaged your provider as a named, automatic, or unnamed module and if you configured it so that the `ServiceLoader` class can search for it.

If you packaged your provider as a named module and have configured it so that the `ServiceLoader` class can search for it (by registering it with its name in the `java.security` as described in Step 8.1: Configure the Provider), then run your test program with the following command:

```
java --module-path "jars" <other java options>
```

The directory `jars` contains your provider.

You may require more options depending on your provider code style (see Step 3.1: Create a Provider That Uses String Objects to Register Its Services and Step 3.2: Create a Provider That Uses Provider.Service), if you packaged your provider in a different kind of module, or if you have not configured it for the `ServiceLoader` class. The following table describes these options.

For the `java` commands, the name of the provider is `MyProvider`, its fully qualified class name is `p.MyProvider`, and it is packaged in the file `com.foo.MyProvider.jar`, which is in the directory `jars`.

Table 3-2 Expected Java Runtime Options for Various Provider Implementation Styles

Module Type	Provider Code Style	Configured for ServiceLoader Class?	Provider Name Used in java.security File	java Command
Unnamed	`String` objects or `Provider.Service`	No	Fully qualified class name	`java -cp "jars/com.foo.MyProvider.jar" <other java options>`
Unnamed	`String` objects or `Provider.Service`	Yes	Fully qualified class name or provider name	`java -cp "jars/com.foo.MyProvider.jar" <other java options>`
Automatic	`String` objects or `Provider.Service`	No	Fully qualified class name	`java --module-path "jars/com.foo.MyProvider.jar" --add-modules=com.foo.MyProvider <other java options>`
Automatic	`String` objects or `Provider.Service`	Yes	Fully qualified class name or provider name	`java --module-path "jars/com.foo.MyProvider.jar" <other java options>`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 3-2 (Cont.) Expected Java Runtime Options for Various Provider Implementation Styles

Module Type	Provider Code Style	Configured for ServiceLoader Class?	Provider Name Used in java.security File	java Command
Named	`String` objects or `Provider.Service`	No	Fully qualified class name	`java --module-path "jars" --add-modules=com.foo.MyProvider --add-exports=com.foo.MyProvider/p=java.base <other java options>` You can remove the `--add-exports` option if you add `exports p` in the module declaration.
Named	`String` objects	Yes	Fully qualified class name	`java --module-path "jars" --add-exports=com.foo.MyProvider/p=java.base <other java options>` You can remove the `--add-exports` option if you add `exports p` in the module declaration.
Named	`String` objects	Yes	Provider name	`java --module-path "jars" --add-exports=com.foo.MyProvider/p=java.base <other java options>` You can remove the `--add-exports` option if you add `exports p` in the module declaration.
Named	`Provider.Service`	Yes	Fully qualified class name	`java --module-path "jars" --add-exports=com.foo.MyProvider/p=java.base<other java options>` You can remove the `--add-exports` option if you add `exports p` in the module declaration.
Named	`Provider.Service`	Yes	Provider name	`java --module-path "jars" <other java options>`

Once you have determined the proper `java` options for your test programs, run them. Debug your code and continue testing as needed. If the Java runtime cannot seem to find one of your algorithms, review the previous steps and ensure that they are all completed.

Be sure to include testing of your programs using different installation options, for example, configured to use the `ServiceLoader` class or to be found in the class path or module path.

1. Optional: If you find during testing that your code needs modification, make the changes and recompile Step 5: Compile Your Code.
2. Place the updated provider code in a JAR file (Step 6: Place Your Provider in a JAR File).
3. Sign the JAR file (Step 7: Sign Your JAR File, If Necessary).
4. Re-configure the provider (Step 8.1: Configure the Provider).
5. Run your programs.
6. Optional: If required, repeat steps 1 to 5.

Step 11: Apply for U.S. Government Export Approval If Required

All U.S. vendors whose providers may be exported outside the U.S. should apply to the Bureau of Industry and Security in the U.S. Department of Commerce for export approval.

Please consult your export counsel for more information.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> If your provider calls `Cipher.getInstance()` and the returned `Cipher` object needs to perform strong cryptography regardless of what cryptographic strength is allowed by the user's downloaded jurisdiction policy files, you should include a copy of the `cryptoPerms` permission policy file which you intend to bundle in the JAR file for your provider and which specifies an appropriate permission for the required cryptographic strength. The necessity for this file is just like the requirement that applications "exempt" from cryptographic restrictions must include a `cryptoPerms` permission policy file in their JAR file. See How to Make Applications Exempt from Cryptographic Restrictions.

Here are two URLs that may be useful:

- US Department of Commerce
- Bureau of Industry and Security

Step 12: Document Your Provider and Its Supported Services

The next step is to write documentation for your clients. At the minimum, you need to specify:

- The name programs should use to refer to your provider.

> **Note**
>
> As of this writing, provider name searches are **case-sensitive**. That is, if your master class specifies your provider name as "CryptoX" but a user requests "CRYPTOx", your provider will not be found. This behavior may change in the future, but for now be sure to warn your clients to use the exact case you specify.

- The types of algorithms and other services implemented by your provider.
- Instructions for installing the provider, similar to those provided in Step 8.1: Configure the Provider, except that the information and examples should be specific to your provider.

In addition, your documentation should specify anything else of interest to clients, such as any default algorithm parameters.

Step 12.1: Indicate Whether Your Implementation is Cloneable for Message Digests and MACs

For each Message Digest and MAC algorithm, indicate whether or not your implementation is cloneable. This is not technically necessary, but it may save clients some time and coding by telling them whether or not intermediate Message Digests or MACs may be possible through cloning.

Clients who do not know whether or not a `MessageDigest` or `Mac` implementation is cloneable can find out by attempting to clone the object and catching the potential exception, as illustrated by the following example:

```
try {
    // try and clone it
    /* compute the MAC for i1 */
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        mac.update(i1);
        byte[] i1Mac = mac.clone().doFinal();

        /* compute the MAC for i1 and i2 */
        mac.update(i2);
        byte[] i12Mac = mac.clone().doFinal();

        /* compute the MAC for i1, i2 and i3 */
        mac.update(i3);
        byte[] i123Mac = mac.doFinal();
    } catch (CloneNotSupportedException cnse) {
        // have to use an approach not involving cloning
    }
```

Where,

`mac`
Indicates the MAC object they received when they requested one via a call to `Mac.getInstance`

`i1`, `i2` and `i3`
Indicates input byte arrays, and they want to calculate separate hashes for:

- `i1`
- `i1 and i2`
- `i1, i2, and i3`

Key Pair Generators

For a key pair generator algorithm, in case the client does not explicitly initialize the key pair generator (via a call to an `initialize` method), each provider must supply and document a default initialization.

For example, the Diffie-Hellman key pair generator supplied by the *SunJCE* provider uses a default prime modulus size (`keysize`) of 2048 bits.

Key Factories

A provider should document all the key specifications supported by its (secret-)key factory.

Algorithm Parameter Generators

In case the client does not explicitly initialize the algorithm parameter generator (via a call to an `init` method in the `AlgorithmParameterGenerator` engine class), each provider must supply and document a default initialization.

For example, the *SunJCE* provider uses a default prime modulus size (`keysize`) of 2048 bits for the generation of Diffie-Hellman parameters, and the *Sun* provider uses a default modulus prime size of 2048 bits for the generation of DSA parameters.

Signature Algorithms

If you implement a signature algorithm, you should document the format in which the signature (generated by one of the `sign` methods) is encoded.

For example, the SHA256withDSA signature algorithm supplied by the "SUN" provider encodes the signature as a standard `ASN.1 SEQUENCE` of two integers, `r` and `s`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Random Number Generation (SecureRandom) Algorithms

For a random number generation algorithm, provide information regarding how "random" the numbers generated are, and the quality of the seed when the random number generator is self-seeding. Also note what happens when a `SecureRandom` object (and its encapsulated `SecureRandomSpi` implementation object) is deserialized: If subsequent calls to the `nextBytes` method (which invokes the `engineNextBytes` method of the encapsulated `SecureRandomSpi` object) of the restored object yield the exact same (random) bytes as the original object would, then let users know that if this behavior is undesirable, they should seed the restored random object by calling its `setSeed` method.

Certificate Factories

A provider should document what types of certificates (and their version numbers, if relevant), can be created by the factory.

Keystores

A provider should document any relevant information regarding the keystore implementation, such as its underlying data format.

Step 13: Make Your Class Files and Documentation Available to Clients

After writing, configuring, testing, installing and documenting your provider software, make documentation available to your customers.

Further Implementation Details and Requirements

This section provides additional information about alias names, service interdependencies, algorithm parameter generators and algorithm parameters.

Alias Names

In the JDK, the aliasing scheme enables clients to use aliases when referring to algorithms or types, rather than the standard names.

For many cryptographic algorithms and types, there is a single official "standard name" defined in the Java Security Standard Algorithm Names.

For example, "SHA-256" is the standard name for the SHA-256 Message Digest algorithm defined in FIPS PUB 180-4: Secure Hash Standard (SHS). `DiffieHellman` is the standard for the Diffie-Hellman key agreement algorithm defined in PKCS#3.

In the JDK, there is an aliasing scheme that enables clients to use aliases when referring to algorithms or types, rather than their standard names.

For example, the "SUN" provider's master class (`Sun.java`) defines the alias `"SHA1/DSA"` for the algorithm whose standard name is `"SHA1withDSA"`. Thus, the following statements are equivalent:

```
Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
Signature sig = Signature.getInstance("SHA1/DSA", "SUN");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Aliases can be defined in your "master class" (see Step 3: Write Your Master Class, a Subclass of Provider). To define an alias, create a property named

```
Alg.Alias.engineClassName.aliasName
```

where *engineClassName* is the name of an engine class (e.g., `Signature`), and *aliasName* is your alias name. The *value* of the property must be the standard algorithm (or type) name for the algorithm (or type) being aliased.

As an example, the "SUN" provider defines the alias `"SHA1/DSA"` for the signature algorithm whose standard name is `"SHA1withDSA"` by setting a property named `Alg.Alias.Signature.SHA1/DSA` to have the value `SHA1withDSA` via the following:

```
put("Alg.Alias.Signature.SHA1/DSA", "SHA1withDSA");
```

> **Note**
>
> The aliases defined by one provider are available only to that provider and not to any other providers. Thus, aliases defined by the *SunJCE* provider are available only to the *SunJCE* provider.

Service Interdependencies

Some algorithms require the use of other types of algorithms. For example, a PBE algorithm usually needs to use a message digest algorithm in order to transform a password into a key.

If you are implementing one type of algorithm that requires another, you can do one of the following:

- Provide your own implementations for both.
- Let your implementation of one algorithm use an instance of the other type of algorithm, as supplied by the default *Sun* provider that is included with every Java SE Platform installation. For example, if you are implementing a PBE algorithm that requires a message digest algorithm, you can obtain an instance of a class implementing the SHA-256 message digest algorithm by calling:

  ```
  MessageDigest.getInstance("SHA-256", "SUN")
  ```

- Let your implementation of one algorithm use an instance of the other type of algorithm, as supplied by another specific provider. This is only appropriate if you are sure that all clients who will use your provider will also have the other provider installed.
- Let your implementation of one algorithm use an instance of the other type of algorithm, as supplied by another (unspecified) provider. That is, you can request an algorithm by name, but without specifying any particular provider, as in:

  ```
  MessageDigest.getInstance("SHA-256")
  ```

  This is only appropriate if you are sure that there will be at least one implementation of the requested algorithm (in this case, SHA-256) installed on each Java platform where your provider will be used.

Here are some common types of algorithm interdependencies:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Signature and Message Digest Algorithms

A signature algorithm often requires use of a message digest algorithm. For example, the SHA256withDSA signature algorithm requires the SHA-256 message digest algorithm.

Signature and (Pseudo-)Random Number Generation Algorithms

A signature algorithm often requires use of a (pseudo-)random number generation algorithm. For example, such an algorithm is required in order to generate a DSA signature.

Key Pair Generation and Message Digest Algorithms

A key pair generation algorithm often requires use of a message digest algorithm. For example, DSA keys are generated using the SHA-256 message digest algorithm.

Algorithm Parameter Generation and Message Digest Algorithms

An algorithm parameter generator often requires use of a message digest algorithm. For example, DSA parameters are generated using the SHA-256 message digest algorithm.

Keystores and Message Digest Algorithms

A keystore implementation will often utilize a message digest algorithm to compute keyed hashes (where the key is a user-provided password) to check the integrity of a keystore and make sure that the keystore has not been tampered with.

Key Pair Generation Algorithms and Algorithm Parameter Generators

A key pair generation algorithm sometimes needs to generate a new set of algorithm parameters. It can either generate the parameters directly, or use an algorithm parameter generator.

Key Pair Generation, Algorithm Parameter Generation, and (Pseudo-)Random Number Generation Algorithms

A key pair generation algorithm may require a source of randomness in order to generate a new key pair and possibly a new set of parameters associated with the keys. That source of randomness is represented by a `SecureRandom` object. The implementation of the key pair generation algorithm may generate the key parameters itself, or may use an algorithm parameter generator to generate them, in which case it may or may not initialize the algorithm parameter generator with a source of randomness.

Algorithm Parameter Generators and Algorithm Parameters

An algorithm parameter generator's `engineGenerateParameters` method must return an `AlgorithmParameters` instance.

Signature and Key Pair Generation Algorithms or Key Factories

If you are implementing a signature algorithm, your implementation's `engineInitSign` and `engineInitVerify` methods will require passed-in keys that are valid for the underlying algorithm (e.g., DSA keys for the DSS algorithm). You can do one of the following:

- Also create your own classes implementing appropriate interfaces (e.g. classes implementing the `DSAPrivateKey` and `DSAPublicKey` interfaces from the package `java.security.interfaces`), and create your own key pair generator and/or key factory returning keys of those types. Require the keys passed to `engineInitSign` and

Copyright © 1993, 2026, Oracle and/or its affiliates.

`engineInitVerify` to be the types of keys you have implemented, that is, keys generated from your key pair generator or key factory. Or you can,

- Accept keys from other key pair generators or other key factories, as long as they are instances of appropriate interfaces that enable your signature implementation to obtain the information it needs (such as the private and public keys and the key parameters). For example, the `engineInitSign` method for a DSS `Signature` class could accept any private keys that are instances of `java.security.interfaces.DSAPrivateKey`.

Keystores and Key and Certificate Factories

A keystore implementation will often utilize a key factory to parse the keys stored in the keystore, and a certificate factory to parse the certificates stored in the keystore.

Default Initialization

In case the client does not explicitly initialize a key pair generator or an algorithm parameter generator, each provider of such a service must supply (and document) a default initialization.

`SUN`JDK Providers Documentation

Default Key Pair Generator Parameter Requirements

If you implement a key pair generator, your implementation should supply default parameters that are used when clients don't specify parameters.

The documentation you supply (Step 12: Document Your Provider and Its Supported Services) should state what the default parameters are.

For example, the DSA key pair generator in the SUN provider supplies a set of precomputed p, q, and g default values for the generation of key pairs in all supported keysizes.

The following example obtains the p, q, and g values for the DSA key pair generator of the preferred provider (as specified in the `java.security` file) as well as the provider's name and the default keysize:

```
static void printDSA() throws Exception {
    KeyPairGenerator kpgen = KeyPairGenerator.getInstance("DSA");
    Provider prov = kpgen.getProvider();
    System.out.println("Current provider: " + prov.getName());

    KeyPair kp = kpgen.genKeyPair();
    DSAPublicKey pubKey = (DSAPublicKey) kp.getPublic();

    DSAParams params = pubKey.getParams();
    BigInteger p = params.getP();
    BigInteger q = params.getQ();
    BigInteger g = params.getG();

    System.out.println("Bit length of p: " + p.bitLength());
    System.out.printf("p: %x\n", p);
    System.out.printf("q: %x\n", q);
    System.out.printf("g: %x\n", g);
}
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

This example prints output similar to the following (line breaks and spaces have been added for clarity):

```
Current provider: SUN
Bit length of p: 2048
p: 8f7935d9 b9aae9bf abed887a cf4951b6 f32ec59e 3baf3718 e8eac496 1f3efd36
   06e74351 a9c41833 39b809e7 c2ae1c53 9ba7475b 85d011ad b8b47987 75498469
   5cac0e8f 14b33608 28a22ffa 27110a3d 62a99345 3409a0fe 696c4658 f84bdd20
   819c3709 a01057b1 95adcd00 233dba54 84b6291f 9d648ef8 83448677 979cec04
   b434a6ac 2e75e998 5de23db0 292fc111 8c9ffa9d 8181e733 8db792b7 30d7b9e3
   49592f68 09987215 3915ea3d 6b8b4653 c633458f 803b32a4 c2e0f272 90256e4e
   3f8a3b08 38a1c450 e4e18c1a 29a37ddf 5ea143de 4b66ff04 903ed5cf 1623e158
   d487c608 e97f211c d81dca23 cb6e3807 65f822e3 42be484c 05763939 601cd667
q: baf696a6 8578f7df dee7fa67 c977c785 ef32b233 bae580c0 bcd5695d
g: 16a65c58 20485070 4e7502a3 9757040d 34da3a34 78c154d4 e4a5c02d 242ee04f
   96e61e4b d0904abd ac8f37ee b1e09f31 82d23c90 43cb642f 88004160 edf9ca09
   b32076a7 9c32a627 f2473e91 879ba2c4 e744bd20 81544cb5 5b802c36 8d1fa83e
   d489e94e 0fa0688e 32428a5c 78c478c6 8d0527b7 1c9a3abb 0b0be12c 44689639
   e7d3ce74 db101a65 aa2b87f6 4c6826db 3ec72f4b 5599834b b4edb02f 7c90e9a4
   96d3a55d 535bebfc 45d4f619 f63f3ded bb873925 c2f224e0 7731296d a887ec1e
   4748f87e fb5fdeb7 5484316b 2232dee5 53ddaf02 112b0d1f 02da3097 3224fe27
   aeda8b9d 4b2922d9 ba8be39e d9e103a6 3c52810b c688b7e2 ed4316e1 ef17dbde
```

The Provider.Service Class

`Provider.Service` class offers an alternative way for providers to advertise their services and supports additional features.

Since its introduction, security providers have published their service information via appropriately formatted key-value String pairs they put in their Hashtable entries. While this mechanism is simple and convenient, it limits the amount customization possible. As a result, JDK 5.0 introduced a second option, the `Provider.Service` class. It offers an alternative way for providers to advertise their services and supports additional features. Note that this addition is fully compatible with the older method of using String valued Hashtable entries. A provider on JDK 5.0 can choose either method as it prefers, or even use both at the same time.

A `Provider.Service` object encapsulates all information about a service. This is the provider that offers the service, its type (e.g. `MessageDigest` or `Signature`), the algorithm name, and the name of the class that implements the service. Optionally, it also includes a list of alternate algorithm names for this service (aliases) and attributes, which are a map of (name, value) String pairs. In addition, it defines the methods `newInstance()` and `supportsParameter()`. They have default implementations, but can be overridden by providers if needed, as may be the case with providers that interface with hardware security tokens.

The `newInstance()` method is used by the security framework when it needs to construct new implementation instances. The default implementation uses reflection to invoke the standard constructor for the respective type of service. For all standard services except `CertStore`, this is the no-args constructor. The `constructorParameter` to `newInstance()` must be null in theses cases. For services of type `CertStore`, the constructor that takes a `CertStoreParameters` object is invoked, and `constructorParameter` must be a non-null instance of `CertStoreParameters`. A security provider can override the `newInstance()` method to implement instantiation as appropriate for that implementation. It could use direct invocation or call a constructor that passes additional information specific to the Provider instance or token. For example, if multiple Smartcard readers are present on the system, it might pass information about which reader the newly created service is to be associated with.

Copyright © 1993, 2026, Oracle and/or its affiliates.

However, despite customization all implementations must follow the conventions about `constructorParameter` described previously.

The `supportsParameter()` tests whether the Service can use the specified parameter. It returns false if this service cannot use the parameter. It returns true if this service can use the parameter, if a fast test is infeasible, or if the status is unknown. It is used by the security framework with some types of services to quickly exclude non-matching implementations from consideration. It is currently only defined for the following standard services: `Signature`, `Cipher`, `Mac`, and `KeyAgreement`. The `parameter` must be an instance of `Key` in these cases. For example, for `Signature` services, the framework tests whether the service can use the supplied Key before instantiating the service. The default implementation examines the attributes `SupportedKeyFormats` and `SupportedKeyClasses`. Again, a provider may override this methods to implement additional tests.

The `SupportedKeyFormats` attribute is a list of the supported formats for encoded keys (as returned by `key.getFormat()`) separated by the "|" (pipe) character. For example, `X.509|PKCS#8`. The `SupportedKeyClasses` attribute is a list of the names of classes of interfaces separated by the "|" character. A key object is considered to be acceptable if it is assignable to at least one of those classes or interfaces named. In other words, if the class of the key object is a subclass of one of the listed classes (or the class itself) or if it implements the listed interface. An example value is `"java.security.interfaces.RSAPrivateKey|java.security.interfaces.RSAPublicKey"` .

Four methods have been added to the Provider class for adding and looking up Services. As mentioned earlier, the implementation of those methods and also of the existing Properties methods have been specifically designed to ensure compatibility with existing Provider subclasses. This is achieved as follows:

If legacy Properties methods are used to add entries, the Provider class makes sure that the property strings are parsed into equivalent Service objects prior to lookup via `getService()`. Similarly, if the `putService()` method is used, equivalent property strings are placed into the provider's hashtable at the same time. If a provider implementation overrides any of the methods in the Provider class, it has to ensure that its implementation does not interfere with this conversion. To avoid problems, we recommend that implementations do not override any of the methods in the `Provider` class.

Signature Formats

The signature algorithm should specify the format in which the signature is encoded.

If you implement a signature algorithm, the documentation you supply (Step 12: Document Your Provider and Its Supported Services) should specify the format in which the signature (generated by one of the `sign` methods) is encoded.

For example, the *SHA1withDSA* signature algorithm supplied by the *Sun* provider encodes the signature as a standard ASN.1 sequence of two `ASN.1 INTEGER` values: `r` and `s`, in that order:

```
SEQUENCE ::= {
        r INTEGER,
        s INTEGER }
```

DSA Interfaces and their Required Implementations

The Java Security API contains interfaces (in the `java.security.interfaces` package) for the convenience of programmers implementing DSA services.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Java Security API contains the following interfaces:

- `DSAKey`
- `DSAKeyPairGenerator`
- `DSAParams`
- `DSAPrivateKey`
- `DSAPublicKey`

The following sections discuss requirements for implementations of these interfaces.

DSAKeyPairGenerator

The interface `DSAKeyPairGenerator` is obsolete. It used to be needed to enable clients to provide DSA-specific parameters to be used rather than the default parameters your implementation supplies. However, it's no longer necessary. The `KeyPairGenerator.initialize` method that takes an `AlgorithmParameterSpec` parameter enables clients to indicate algorithm-specific parameters.

DSAParams Implementation

If you are implementing a DSA key pair generator, you need a class implementing `DSAParams` for holding and returning the p, q, and g parameters.

A `DSAParams` implementation is also required if you implement the `DSAPrivateKey` and `DSAPublicKey` interfaces. `DSAPublicKey` and `DSAPrivateKey` both extend the DSAKey interface, which contains a `getParams` method that must return a `DSAParams` object.

> **Note**
>
> There is a `DSAParams` implementation built into the JDK: the `java.security.spec.DSAParameterSpec` class.

DSAPrivateKey and DSAPublicKey Implementations

If you implement a DSA key pair generator or key factory, you need to create classes implementing the `DSAPrivateKey` and `DSAPublicKey` interfaces.

If you implement a DSA key pair generator, your `generateKeyPair` method (in your `KeyPairGeneratorSpi` subclass) will return instances of your implementations of those interfaces.

If you implement a DSA key factory, your `engineGeneratePrivate` method (in your `KeyFactorySpi` subclass) will return an instance of your `DSAPrivateKey` implementation, and your `engineGeneratePublic` method will return an instance of your `DSAPublicKey` implementation.

Also, your `engineGetKeySpec` and `engineTranslateKey` methods will expect the passed-in key to be an instance of a `DSAPrivateKey` or `DSAPublicKey` implementation. The `getParams` method provided by the interface implementations is useful for obtaining and extracting the parameters from the keys and then using the parameters, for example as parameters to the `DSAParameterSpec` constructor called to create a parameter specification from parameter values that could be used to initialize a `KeyPairGenerator` object for DSA.

Copyright © 1993, 2026, Oracle and/or its affiliates.

If you implement a DSA signature algorithm, your `engineInitSign` method (in your `SignatureSpi` subclass) will expect to be passed a `DSAPrivateKey` and your `engineInitVerify` method will expect to be passed a `DSAPublicKey`.

Please note: The `DSAPublicKey` and `DSAPrivateKey` interfaces define a very generic, provider-independent interface to DSA public and private keys, respectively. The `engineGetKeySpec` and `engineTranslateKey` methods (in your `KeyFactorySpi` subclass) could additionally check if the passed-in key is actually an instance of their provider's own implementation of `DSAPrivateKey` or `DSAPublicKey`, for example, to take advantage of provider-specific implementation details. The same is true for the DSA signature algorithm `engineInitSign` and `engineInitVerify` methods (in your `SignatureSpi` subclass).

To see what methods need to be implemented by classes that implement the `DSAPublicKey` and `DSAPrivateKey` interfaces, first note the following interface signatures:

In the `java.security.interfaces` package:

```
public interface DSAPrivateKey extends DSAKey, PrivateKey
public interface DSAPublicKey extends DSAKey, PublicKey
public interface DSAKey
```

In the `java.security` package:

```
public interface PrivateKey extends Key
public interface PublicKey extends Key
public interface Key extends Serializable
```

To implement the `DSAPrivateKey` and `DSAPublicKey` interfaces, you must implement the methods they define as well as those defined by interfaces they extend, directly or indirectly.

Thus, for private keys, you need to supply a class that implements:

- The `getX` method from the `DSAPrivateKey` interface.
- The `getParams` method from the `DSAKey` interface because `DSAPrivateKey` extends `DSAKey`.

> **Note**
>
> The `getParams` method returns a `DSAParams` object, so you must also have a `DSAParams` implementation.

- The `getAlgorithm`, `getEncoded`, and `getFormat` methods from the `Key` interface because `DSAPrivateKey` extends `java.security.PrivateKey`, and `PrivateKey` extends `Key`.

Similarly, for public DSA keys, you need to supply a class that implements:

- The `getY` method from the `DSAPublicKey` interface.
- The `getParams` method from the `DSAKey` interface because `DSAPublicKey` extends DSAKey.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The `getParams` method returns a `DSAParams` object, so you must also have a `DSAParams` implementation.

- The `getAlgorithm`, `getEncoded`, and `getFormat` methods from the `Key` interface because `DSAPublicKey` extends `java.security.PublicKey`, and `PublicKey` extends `Key`.

RSA Interfaces and their Required Implementations

The Java Security API contains the interfaces (in the `java.security.interfaces` package) for the convenience of programmers implementing RSA services.

- `RSAPrivateKey`
- `RSAPrivateCrtKey`
- `RSAPublicKey`

The following sections discuss requirements for implementations of these interfaces.

RSAPrivateKey, RSAPrivateCrtKey, and RSAPublicKey Implementations

If you implement an RSA key pair generator or key factory, you need to create classes implementing the `RSAPublicKey` (and/or `RSAPrivateCrtKey`) and `RSAPublicKey` interfaces. (`RSAPrivateCrtKey` is the interface to an RSA private key, using the *Chinese Remainder Theorem* (CRT) representation.)

If you implement an RSA key pair generator, your `generateKeyPair` method (in your `KeyPairGeneratorSpi` subclass) will return instances of your implementations of those interfaces.

If you implement an RSA key factory, your `engineGeneratePrivate` method (in your `KeyFactorySpi` subclass) will return an instance of your `RSAPrivateKey` (or `RSAPrivateCrtKey`) implementation, and your `engineGeneratePublic` method will return an instance of your `RSAPublicKey` implementation.

Also, your `engineGetKeySpec` and `engineTranslateKey` methods will expect the passed-in key to be an instance of an `RSAPrivateKey`, `RSAPrivateCrtKey`, or `RSAPublicKey` implementation.

If you implement an RSA signature algorithm, your `engineInitSign` method (in your `SignatureSpi` subclass) will expect to be passed either an `RSAPrivateKey` or an `RSAPrivateCrtKey`, and your `engineInitVerify` method will expect to be passed an `RSAPublicKey`.

Please note: The `RSAPublicKey`, `RSAPrivateKey`, and `RSAPrivateCrtKey` interfaces define a very generic, provider-independent interface to RSA public and private keys. The `engineGetKeySpec` and `engineTranslateKey` methods (in your `KeyFactorySpi` subclass) could additionally check if the passed-in key is actually an instance of their provider's own implementation of `RSAPrivateKey`, `RSAPrivateCrtKey`, or `RSAPublicKey`, for example, to take advantage of provider-specific implementation details. The same is true for the RSA signature algorithm `engineInitSign` and `engineInitVerify` methods (in your `SignatureSpi` subclass).

To see what methods need to be implemented by classes that implement the `RSAPublicKey`, `RSAPrivateKey`, and `RSAPrivateCrtKey` interfaces, first note the following interface signatures:

Copyright © 1993, 2026, Oracle and/or its affiliates.

In the `java.security.interfaces` package:

```
public interface RSAPrivateKey extends PrivateKey
public interface RSAPrivateCrtKey extends RSAPrivateKey
public interface RSAPublicKey extends PublicKey
```

In the `java.security` package:

```
public interface PrivateKey extends Key
public interface PublicKey extends Key
public interface Key extends Serializable
```

To implement the `RSAPrivateKey`, `RSAPrivateCrtKey`, and `RSAPublicKey` interfaces, you must implement the methods they define as well as those defined by interfaces they extend, directly or indirectly.

Thus, for RSA private keys, you need to supply a class that implements:

- The `getModulus` and `getPrivateExponent` methods from the `RSAPrivateKey` interface.
- The `getAlgorithm`, `getEncoded`, and `getFormat` methods from the `Key` interface because `RSAPrivateKey` extends `java.security.PrivateKey`, and `PrivateKey` extends `Key`.

Similarly, for RSA private keys using the *Chinese Remainder Theorem* (CRT) representation, you need to supply a class that implements:

- All the methods listed previously for RSA private keys because `RSAPrivateCrtKey` extends `java.security.interfaces.RSAPrivateKey`.
- The `getPublicExponent`, `getPrimeP`, `getPrimeQ`, `getPrimeExponentP`, `getPrimeExponentQ`, and `getCrtCoefficient` methods from the `RSAPrivateKey` interface.

For public RSA keys, you need to supply a class that implements:

- The `getModulus` and `getPublicExponent` methods from the `RSAPublicKey` interface.
- The `getAlgorithm`, `getEncoded`, and `getFormat` methods from the `Key` interface because `RSAPublicKey` extends `java.security.PublicKey`, and `PublicKey` extends `Key`.

JCA contains a number of `AlgorithmParameterSpec` implementations for the most frequently used cipher and key agreement algorithm parameters. If you are operating on algorithm parameters that should be for a different type of algorithm not provided by JCA, you will need to supply your own `AlgorithmParameterSpec` implementation appropriate for that type of algorithm.

Diffie-Hellman Interfaces and their Required Implementations

JCA contains interfaces (in the `javax.crypto.interfaces` package) for the convenience of programmers implementing Diffie-Hellman services.

- `DHPublicKey`
- `DHKey`
- `DHPrivateKey`

The following sections discuss requirements for implementations of these interfaces.

Copyright © 1993, 2026, Oracle and/or its affiliates.

DHPrivateKey and DHPublicKey Implementations

If you implement a Diffie-Hellman key pair generator or key factory, you need to create classes implementing the `DHPrivateKey` and `DHPublicKey` interfaces.

If you implement a Diffie-Hellman key pair generator, your `generateKeyPair` method (in your `KeyPairGeneratorSpi` subclass) will return instances of your implementations of those interfaces.

If you implement a Diffie-Hellman key factory, your `engineGeneratePrivate` method (in your `KeyFactorySpi` subclass) will return an instance of your `DHPrivateKey` implementation, and your `engineGeneratePublic` method will return an instance of your `DHPublicKey` implementation.

Also, your `engineGetKeySpec` and `engineTranslateKey` methods will expect the passed-in key to be an instance of a `DHPrivateKey` or `DHPublicKey` implementation. The `getParams` method provided by the interface implementations is useful for obtaining and extracting the parameters from the keys. You can then use the parameters, for example, as parameters to the `DHParameterSpec` constructor called to create a parameter specification from parameter values used to initialize a `KeyPairGenerator` object for Diffie-Hellman.

If you implement the Diffie-Hellman key agreement algorithm, your `engineInit` method (in your `KeyAgreementSpi` subclass) will expect to be passed a `DHPrivateKey` and your `engineDoPhase` method will expect to be passed a `DHPublicKey`.

> **Note**
>
> The `DHPublicKey` and `DHPrivateKey` interfaces define a very generic, provider-independent interface to Diffie-Hellman public and private keys, respectively. The `engineGetKeySpec` and `engineTranslateKey` methods (in your `KeyFactorySpi` subclass) could additionally check if the passed-in key is actually an instance of their provider's own implementation of `DHPrivateKey` or `DHPublicKey`, for example, to take advantage of provider-specific implementation details. The same is true for the Diffie-Hellman algorithm `engineInit` and `engineDoPhase` methods (in your `KeyAgreementSpi` subclass).

To see what methods need to be implemented by classes that implement the `DHPublicKey` and `DHPrivateKey` interfaces, first note the following interface signatures:

In the `javax.crypto.interfaces` package:

```
public interface DHPrivateKey extends DHKey, PrivateKey
public interface DHPublicKey extends DHKey, jPublicKey
public interface DHKey
```

In the `java.security` package:

```
public interface PrivateKey extends Key
public interface PublicKey extends Key
public interface Key extends Serializable
```

To implement the `DHPrivateKey` and `DHPublicKey` interfaces, you must implement the methods they define as well as those defined by interfaces they extend, directly or indirectly.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Thus, for private keys, you need to supply a class that implements:

- The `getX` method from the `DHPrivateKey` interface.
- The `getParams` method from the `DHKey` interface because `DHPrivateKey` extends `DHKey`.
- The `getAlgorithm`, `getEncoded`, and `getFormat` methods from the `Key` interface because `DHPrivateKey` extends `java.security.PrivateKey`, and `PrivateKey` extends `Key`.

Similarly, for public Diffie-Hellman keys, you need to supply a class that implements:

- The `getY` method from the `DHPublicKey` interface.
- The `getParams` method from the `DHKey` interface because `DHPublicKey` extends `DHKey`.
- The `getAlgorithm`, `getEncoded`, and `getFormat` methods from the `Key` interface because `DHPublicKey` extends `java.security.PublicKey`, and `PublicKey` extends `Key`.

Interfaces for Other Algorithm Types

As noted previously, the Java Security API contains interfaces for the convenience of programmers implementing services like DSA, RSA and ECC. If there are services without API support, you need to define your own APIs.

If you are implementing a key pair generator for a different algorithm, you should create an interface with one or more `initialize` methods that clients can call when they want to provide algorithm-specific parameters to be used rather than the default parameters your implementation supplies. Your subclass of `KeyPairGeneratorSpi` should implement this interface.

For algorithms without direct API support, it is recommended that you create similar interfaces and provide implementation classes. Your public key interface should extend the `PublicKey` interface. Similarly, your private key interface should extend the `PrivateKey` interface.

Algorithm Parameter Specification Interfaces and Classes

An algorithm parameter specification is a transparent representation of the sets of parameters used with an algorithm.

A *transparent* representation of parameters means that you can access each value individually, through one of the *get* methods defined in the corresponding specification class (e.g., `DSAParameterSpec` defines `getP`, `getQ`, and `getG` methods, to access the p, q, and g parameters, respectively).

This is contrasted with an *opaque* representation, as supplied by the `AlgorithmParameters` engine class, in which you have no direct access to the key material values; you can only get the name of the algorithm associated with the parameter set (via `getAlgorithm`) and some kind of encoding for the parameter set (via `getEncoded`).

If you supply an `AlgorithmParametersSpi`, `AlgorithmParameterGeneratorSpi`, or `KeyPairGeneratorSpi` implementation, you must utilize the `AlgorithmParameterSpec` interface, since each of those classes contain methods that take an `AlgorithmParameterSpec` parameter. Such methods need to determine which actual implementation of that interface has been passed in, and act accordingly.

JCA contains a number of `AlgorithmParameterSpec` implementations for the most frequently used signature, cipher and key agreement algorithm parameters. If you are operating on algorithm parameters that should be for a different type of algorithm not provided by JCA, you will need to supply your own `AlgorithmParameterSpec` implementation appropriate for that type of algorithm.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Java defines the following algorithm parameter specification interfaces and classes in the `java.security.spec` and `javax.crypto.spec` packages:

The AlgorithmParameterSpec Interface

`AlgorithmParameterSpec` is an interface to a transparent specification of cryptographic parameters.

This interface contains no methods or constants. Its only purpose is to group (and provide type safety for) all parameter specifications. All parameter specifications must implement this interface.

The DSAParameterSpec Class

This class (which implements the `AlgorithmParameterSpec` and `DSAParams` interfaces) specifies the set of parameters used with the DSA algorithm. It has the following methods:

```
public BigInteger getP()

public BigInteger getQ()

public BigInteger getG()
```

These methods return the DSA algorithm parameters: the prime `p`, the sub-prime `q`, and the base `g`.

Many types of DSA services will find this class useful - for example, it is utilized by the DSA signature, key pair generator, algorithm parameter generator, and algorithm parameters classes implemented by the *Sun* provider. As a specific example, an algorithm parameters implementation must include an implementation for the `getParameterSpec` method, which returns an `AlgorithmParameterSpec`. The DSA algorithm parameters implementation supplied by *Sun* returns an instance of the `DSAParameterSpec` class.

The IvParameterSpec Class

This class (which implements the `AlgorithmParameterSpec` interface) specifies the initialization vector (IV) used with a cipher in feedback mode.

Table 3-3 Method in `IvParameterSpec`

Method	Description
`byte[] getIV()`	Returns the initialization vector (IV).

The OAEPParameterSpec Class

This class specifies the set of parameters used with OAEP Padding, as defined in the PKCS #1 standard.

Table 3-4 Methods in `OAEPParameterSpec`

Method	Description
`String getDigestAlgorithm()`	Returns the message digest algorithm name.
`String getMGFAlgorithm()`	Returns the mask generation function algorithm name.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 3-4 (Cont.) Methods in `OAEPParameterSpec`

Method	Description
`AlgorithmParameterSpec getMGFParameters()`	Returns the parameters for the mask generation function.
`PSource getPSource()`	Returns the source of encoding input P.

The PBEParameterSpec Class

This class (which implements the `AlgorithmParameterSpec` interface) specifies the set of parameters used with a password-based encryption (PBE) algorithm.

Table 3-5 Methods in `PBEParameterSpec`

Method	Description
`int getIterationCount()`	Returns the iteration count.
`byte[] getSalt()`	Returns the salt.

The RC2ParameterSpec Class

This class (which implements the `AlgorithmParameterSpec` interface) specifies the set of parameters used with the RC2 algorithm.

Table 3-6 Methods in `RC2ParameterSpec`

Method	Description
`boolean equals(Object obj)`	Tests for equality between the specified object and this object.
`int getEffectiveKeyBits()`	Returns the effective key size in bits.
`byte[] getIV()`	Returns the IV or null if this parameter set does not contain an IV.
`int hashCode()`	Calculates a hash code value for the object.

The RC5ParameterSpec Class

This class (which implements the `AlgorithmParameterSpec` interface) specifies the set of parameters used with the RC5 algorithm.

Table 3-7 Methods in `RC5ParameterSpec`

Method	Description
`boolean equals(Object obj)`	Tests for equality between the specified object and this object.
`byte[] getIV()`	Returns the IV or null if this parameter set does not contain an IV.
`int getRounds()`	Returns the number of rounds.
`int getVersion()`	Returns the version.
`int getWordSize()`	Returns the word size in bits.
`int hashCode()`	Calculates a hash code value for the object.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The DHParameterSpec Class

This class (which implements the `AlgorithmParameterSpec` interface) specifies the set of parameters used with the Diffie-Hellman algorithm.

Table 3-8 Methods in DHParameterSpec

Method	Description
`BigInteger getG()`	Returns the base generator `g`.
`int getL()`	Returns the size in bits, `l`, of the random exponent (private value).
`BigInteger getP()`	Returns the prime modulus `p`.

Many types of Diffie-Hellman services will find this class useful; for example, it is used by the Diffie-Hellman key agreement, key pair generator, algorithm parameter generator, and algorithm parameters classes implemented by the "SunJCE" provider. As a specific example, an algorithm parameters implementation must include an implementation for the `getParameterSpec` method, which returns an `AlgorithmParameterSpec`. The Diffie-Hellman algorithm parameters implementation supplied by "SunJCE" returns an instance of the `DHParameterSpec` class.

Key Specification Interfaces and Classes Required by Key Factories

A key factory provides bi-directional conversions between opaque keys (of type `Key`) and key specifications. If you implement a key factory, you thus need to understand and utilize key specifications. In some cases, you also need to implement your own key specifications.

Key specifications are transparent representations of the key material that constitutes a key. If the key is stored on a hardware device, its specification may contain information that helps identify the key on the device.

A *transparent* representation of keys means that you can access each key material value individually, through one of the *get* methods defined in the corresponding specification class. For example, `java.security.spec.DSAPrivateKeySpec` defines `getX`, `getP`, `getQ`, and `getG` methods, to access the private key `x`, and the DSA algorithm parameters used to calculate the key: the prime `p`, the sub-prime `q`, and the base `g`.

This is contrasted with an *opaque* representation, as defined by the Key interface, in which you have no direct access to the parameter fields. In other words, an "opaque" representation gives you limited access to the key - just the three methods defined by the Key interface: `getAlgorithm`, `getFormat`, and `getEncoded`.

A key may be specified in an algorithm-specific way, or in an algorithm-independent encoding format (such as ASN.1). For example, a DSA private key may be specified by its components *x*, *p*, *q*, and *g* (see `DSAPrivateKeySpec`), or it may be specified using its DER encoding (see `PKCS8EncodedKeySpec`).

Java defines the following key specification interfaces and classes in the `java.security.spec` and `javax.crypto.spec` packages:

The KeySpec Interface

This interface contains no methods or constants. Its only purpose is to group (and provide type safety for) all key specifications. All key specifications must implement this interface.

Java supplies several classes implementing the `KeySpec` interface:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- DSAPrivateKeySpec
- DSAPublicKeySpec
- RSAPrivateKeySpec
- RSAPublicKeySpec
- EncodedKeySpec
- PKCS8EncodedKeySpec
- X509EncodedKeySpec

If your provider uses key types (e.g., `Your_PublicKey_type` and `Your_PrivateKey_type`) for which the JDK does not already provide corresponding `KeySpec` classes, there are two possible scenarios, one of which requires that you implement your own key specifications:

1. If your users will never have to access specific key material values of your key type, you will not have to provide any `KeySpec` classes for your key type.

   In this scenario, your users will always create `Your_PublicKey_type` and `Your_PrivateKey_type` keys through the appropriate `KeyPairGenerator` supplied by your provider for that key type. If they want to store the generated keys for later usage, they retrieve the keys' encodings (using the `getEncoded` method of the `Key` interface). When they want to create an `Your_PublicKey_type` or `Your_PrivateKey_type` key from the encoding (e.g., in order to initialize a Signature object for signing or verification), they create an instance of `X509EncodedKeySpec` or `PKCS8EncodedKeySpec` from the encoding, and feed it to the appropriate `KeyFactory` supplied by your provider for that algorithm, whose `generatePublic` and `generatePrivate` methods will return the requested `PublicKey` (an instance of `Your_PublicKey_type`) or `PrivateKey` (an instance of `Your_PrivateKey_type`) object, respectively.

2. If you anticipate a need for users to access specific key material values of your key type, or to construct a key of your key type from key material and associated parameter values, rather than from its encoding (as in the previous case), you have to specify new `KeySpec` classes (classes that implement the `KeySpec` interface) with the appropriate constructor methods and *get* methods for returning key material fields and associated parameter values for your key type. You will specify those classes in a similar manner as is done by the `DSAPrivateKeySpec` and `DSAPublicKeySpec` classes. You need to ship those classes along with your provider classes, for example, as part of your provider JAR file.

The DSAPrivateKeySpec Class

This class (which implements the `KeySpec` Interface) specifies a DSA private key with its associated parameters. It has the following methods:

Table 3-9 Methods in DSAPrivateKeySpec

Method in DSAPrivateKeySpec	Description
`public BigInteger getX()`	Returns the private key x.
`public BigInteger getP()`	Returns the prime p.
`public BigInteger getQ()`	Returns the sub-prime q.
`public BigInteger getG()`	Returns the base g.

These methods return the private key `x`, and the DSA algorithm parameters used to calculate the key: the prime `p`, the sub-prime `q`, and the base `g`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The DSAPublicKeySpec Class

This class (which implements the `KeySpec` Interface) specifies a DSA public key with its associated parameters. It has the following methods:

Table 3-10 Methods in DSAPublicKeySpec

Method in DSAPublicKeySpec	Description
`public BigInteger getY()`	returns the public key y.
`public BigInteger getP()`	Returns the prime p.
`public BigInteger getQ()`	Returns the sub-prime q.
`public BigInteger getG()`	Returns the base g.

The RSAPrivateKeySpec Class

This class (which implements the `KeySpec` Interface) specifies an RSA private key. It has the following methods:

Table 3-11 Methods in RSAPrivateKeySpec

Method in RSAPrivateKeySpec	Description
`public BigInteger getModulus()`	Returns the modulus.
`public BigInteger getPrivateExponent()`	Returns the private exponent.

These methods return the RSA modulus `n` and private exponent `d` values that constitute the RSA private key.

The RSAPrivateCrtKeySpec Class

This class (which extends the `RSAPrivateKeySpec` class) specifies an RSA private key, as defined in the PKCS#1 standard, using the *Chinese Remainder Theorem* (CRT) information values. It has the following methods (in addition to the methods inherited from its superclass `RSAPrivateKeySpec`):

Table 3-12 Methods in RSAPrivateCrtKeySpec

Method in RSAPrivateCrtKeySpec	Description
`public BigInteger getPublicExponent()`	Returns the public exponent.
`public BigInteger getPrimeP()`	Returns the prime P.
`public BigInteger getPrimeQ()`	Returns the prime Q.
`public BigInteger getPrimeExponentP()`	Returns the primeExponentP.
`public BigInteger getPrimeExponentQ()`	Returns the primeExponentQ.
`public BigInteger getCrtCoefficient()`	Returns the crtCoefficient.

These methods return the public exponent `e` and the CRT information integers: the prime factor `p` of the modulus `n`, the prime factor `q` of `n`, the exponent `d mod (p-1)`, the exponent `d mod (q-1)`, and the Chinese Remainder Theorem coefficient `(inverse of q) mod p`.

An RSA private key logically consists of only the modulus and the private exponent. The presence of the CRT values is intended for efficiency.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The RSAPublicKeySpec Class

This class (which implements the `KeySpec` Interface) specifies an RSA public key. It has the following methods:

Table 3-13 Methods in RSAPublicKeySpec

Method in RSAPublicKeySpec	Description
`public BigInteger getModulus()`	Returns the modulus.
`public BigInteger getPublicExponent()`	Returns the public exponent.

The EncodedKeySpec Class

This abstract class (which implements the `KeySpec` Interface) represents a public or private key in encoded format.

Table 3-14 Methods in EncodedKeySpec

Method in EncodedKeySpec	Description
`public abstract byte[] getEncoded()`	Returns the encoded key.
`public abstract String getFormat()`	Returns the name of the encoding format.

The JDK supplies two classes implementing the `EncodedKeySpec` interface: `PKCS8EncodedKeySpec` and `X509EncodedKeySpec`. If desired, you can supply your own `EncodedKeySpec` implementations for those or other types of key encodings.

The PKCS8EncodedKeySpec Class

This class, which is a subclass of `EncodedKeySpec`, represents the DER encoding of a private key, according to the format specified in the PKCS #8 standard.

Its `getEncoded` method returns the key bytes, encoded according to the PKCS #8 standard. Its `getFormat` method returns the string "PKCS#8".

The X509EncodedKeySpec Class

This class, which is a subclass of `EncodedKeySpec`, represents the DER encoding of a public or private key, according to the format specified in the X.509 standard.

Its `getEncoded` method returns the key bytes, encoded according to the X.509 standard. Its `getFormat` method returns the string "X.509".`DHPrivateKeySpec`, `DHPublicKeySpec`, `DESKeySpec`, `DESedeKeySpec`, `PBEKeySpec`, and `SecretKeySpec`.

The DHPrivateKeySpec Class

This class (which implements the `KeySpec` interface) specifies a Diffie-Hellman private key with its associated parameters.

Table 3-15 Methods in DHPrviateKeySpec

Method in DHPrivateKeySpec	Description
`BigInteger getG()`	Returns the base generator `g`.
`BigInteger getP()`	Returns the prime modulus `p`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 3-15 (Cont.) Methods in DHPrviateKeySpec

Method in DHPrivateKeySpec	Description
`BigInteger getX()`	Returns the private value `x`.

The DHPublicKeySpec Class

Table 3-16 Methods in DHPublicKeySpec

Method in DHPublicKeySpec	Description
`BigInteger getG()`	Returns the base generator `g`.
`BigInteger getP()`	Returns the prime modulus `p`.
`BigInteger getY()`	Returns the public value `y`.

The DESKeySpec Class

This class (which implements the `KeySpec` interface) specifies a DES key.

Table 3-17 Methods in DESKeySpec

Method in DESKeySpec	Description
`byte[] getKey()`	Returns the DES key bytes.
`static boolean isParityAdjusted(byte[] key, int offset)`	Checks if the given DES key material is parity-adjusted.
`static boolean isWeak(byte[] key, int offset)`	Checks if the given DES key material is weak or semi-weak.

The DESedeKeySpec Class

This class (which implements the `KeySpec` interface) specifies a DES-EDE (Triple DES) key.

Table 3-18 Methods in DESedeKeySpec

Method in DESedeKeySpec	Description
`byte[] getKey()`	Returns the DES-EDE key.
`static boolean isParityAdjusted(byte[] key, int offset)`	Checks if the given DES-EDE key is parity-adjusted.

The PBEKeySpec Class

This class implements the `KeySpec` interface. A user-chosen password can be used with password-based encryption (PBE); the password can be viewed as a type of raw key material. An encryption mechanism that uses this class can derive a cryptographic key from the raw key material.

Table 3-19 Methods in PBEKeySpec

Method in PBEKeySpec	Description
`void clearPassword`	Clears the internal copy of the password.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 3-19 (Cont.) Methods in PBEKeySpec

Method in PBEKeySpec	Description
`int getIterationCount`	Returns the iteration count or 0 if not specified.
`int getKeyLength`	Returns the to-be-derived key length or 0 if not specified.
`char[] getPassword`	Returns a copy of the password.
`byte[] getSalt`	Returns a copy of the salt or null if not specified.

The SecretKeySpec Class

This class implements the `KeySpec` interface. Since it also implements the `SecretKey` interface, it can be used to construct a `SecretKey` object in a provider-independent fashion, i.e., without having to go through a provider-based `SecretKeyFactory`.

Table 3-20 Methods in SecretKeySpec

Method in SecretKeySpec	Description
`boolean equals (Object obj)`	Indicates whether some other object is "equal to" this one.
`String getAlgorithm()`	Returns the name of the algorithm associated with this secret key.
`byte[] getEncoded()`	Returns the key material of this secret key.
`String getFormat()`	Returns the name of the encoding format for this secret key.
`int hashCode()`	Calculates a hash code value for the object.

Secret-Key Generation

If you provide a secret-key generator (subclass of `javax.crypto.KeyGeneratorSpi`) for a particular secret-key algorithm, you may return the generated secret-key object.

The generated secret-key object (which must be an instance of `javax.crypto.SecretKey`, see engineGenerateKey) can be returned in one of the following ways:

- You implement a class whose instances represent secret-keys of the algorithm associated with your key generator. Your key generator implementation returns instances of that class. This approach is useful if the keys generated by your key generator have provider-specific properties.
- Your key generator returns an instance of `SecretKeySpec`, which already implements the `javax.crypto.SecretKey` interface. You pass the (raw) key bytes and the name of the secret-key algorithm associated with your key generator to the `SecretKeySpec` constructor. This approach is useful if the underlying (raw) key bytes can be represented as a byte array and have no key-parameters associated with them.

Adding New Object Identifiers

The following information applies to providers who supply an algorithm that is not listed as one of the standard algorithms in Java Security Standard Algorithm Names.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Mapping from OID to Name

Sometimes the JCA needs to instantiate a cryptographic algorithm implementation from an algorithm identifier (for example, as encoded in a certificate), which by definition includes the object identifier (OID) of the algorithm. For example, in order to verify the signature on an X.509 certificate, the JCA determines the signature algorithm from the signature algorithm identifier that is encoded in the certificate, instantiates a `Signature` object for that algorithm, and initializes the `Signature` object for verification.

For the JCA to find your algorithm, you should provide the object identifier of your algorithm as an alias entry for your algorithm in the provider master file.

```
put("Alg.Alias.<engine_type>.1.2.3.4.5.6.7.8",
    "<algorithm_alias_name>");
```

Note that if your algorithm is known under more than one object identifier, you need to create an alias entry for each object identifier under which it is known.

An example of where the JCA needs to perform this type of mapping is when your algorithm ("`Foo`") is a signature algorithm and users run the `keytool` command and specify your (signature) algorithm alias.

```
% keytool -genkeypair -sigalg 1.2.3.4.5.6.7.8 -keyalg foo
```

In this case, your provider master file should contain the following entries:

```
put("Signature.Foo", "com.xyz.MyFooSignatureImpl");
put("Alg.Alias.Signature.1.2.3.4.5.6.7.8", "Foo");
put("KeyPairGenerator.Foo", "com.xyz.MyFooKeyPairGeneratorImpl");
```

Other examples of where this type of mapping is performed are (1) when your algorithm is a keytype algorithm and your program parses a certificate (using the X.509 implementation of the SUN provider) and extracts the public key from the certificate in order to initialize a Signature object for verification, and (2) when `keytool` users try to access a private key of your keytype (for example, to perform a digital signature) after having generated the corresponding keypair. In these cases, your provider master file should contain the following entries:

```
put("KeyFactory.Foo", "com.xyz.MyFooKeyFactoryImpl");
put("Alg.Alias.KeyFactory.1.2.3.4.5.6.7.8", "Foo");
```

Mapping from Name to OID

If the JCA needs to perform the inverse mapping (that is, from your algorithm name to its associated OID), you need to provide an alias entry of the following form for one of the OIDs under which your algorithm should be known:

```
put("Alg.Alias.Signature.OID.1.2.3.4.5.6.7.8", "MySigAlg");
```

If your algorithm is known under more than one object identifier, prefix the preferred one with "OID."

An example of where the JCA needs to perform this kind of mapping is when users run `keytool` in any mode that takes a `-sigalg` option. For example, when the `-genkeypair` and `-`

Copyright © 1993, 2026, Oracle and/or its affiliates.

`certreq` commands are invoked, the user can specify your (signature) algorithm with the `-sigalg` option.

Ensuring Exportability

A key feature of JCA is the exportability of the JCA framework and of the provider cryptography implementations if certain conditions are met.

By default, an application can use cryptographic algorithms of any strength. However, due to import regulations in some countries, you may have to limit those algorithms' strength. You do this with jurisdiction policy files; see Cryptographic Strength Configuration. The JCA framework will enforce the restrictions specified in the installed jurisdiction policy files.

As noted elsewhere, you can write just one version of your provider software, implementing cryptography of maximum strength. It is up to JCA, not your provider, to enforce any jurisdiction policy file-mandated restrictions regarding the cryptographic algorithms and maximum cryptographic strengths available to applets/applications in different locations.

The conditions that must be met by your provider in order to enable it to be plugged into JCA are the following:

- The provider code should be written in such a way that provider classes become unusable if instantiated by an application directly, bypassing JCA. See Step 1: Write your Service Implementation Code in Steps to Implement and Integrate a Provider.
- The provider package must be signed by an entity trusted by the JCA framework. (See Step 7.1: Get a Code-Signing Certificate through Step 7.2: Sign Your Provider.) U.S. vendors whose providers may be exported outside the U.S. first need to apply for U.S. government export approval. (See Step 11: Apply for U.S. Government Export Approval If Required.)

Sample Code for MyProvider

The following is the complete source code for an example provider, MyProvider. It's a portable provider; you can specify it in a class or module path. It consists of two modules:

- `com.example.MyProvider`: Contains an example provider that demonstrate how to write a provider with the `Provider.Service` mechanism. You must compile, package, and sign the provider, then specify it in your class or module path as described in Steps to Implement and Integrate a Provider.
- `com.example.MyApp`: Contains a sample application that uses the MyProvider provider. It finds and loads this provider with the `ServiceLoader` mechanism, and then registers it dynamically with the `Security.addProvider()` method.

This example consists of the following files:

- src/com.example.MyProvider/module-info.java
- src/com.example.MyProvider/com/example/MyProvider/MyProvider.java
- src/com.example.MyProvider/com/example/MyProvider/MyCipher.java
- src/com.example.MyProvider/META-INF/services/java.security.Provider
- src/com.example.MyApp/module-info.java
- src/com.example.MyApp/com/example/MyApp/MyApp.java
- RunTest.sh

Copyright © 1993, 2026, Oracle and/or its affiliates.

src/com.example.MyProvider/module-info.java

See Step 4: Create a Module Declaration for Your Provider for information about the module declaration, which is specified in `module-info.java`.

```
module com.example.MyProvider {
    provides java.security.Provider with com.example.MyProvider.MyProvider;
}
```

src/com.example.MyProvider/com/example/MyProvider/MyProvider.java

The `MyProvider` class is an example of a provider that uses the `Provider.Service` class. See Step 3.2: Create a Provider That Uses Provider.Service.

```
package com.example.MyProvider;

import java.security.*;
import java.util.*;

/**
 * Test JCE provider.
 *
 * Registers services using Provider.Service and overrides newInstance().
 */
public final class MyProvider extends Provider {

    public MyProvider() {
        super("MyProvider", "1.0", "My JCE provider");

        final Provider p = this;

        putService(
            new ProviderService(
                p, "Cipher", "MyCipher", "com.example.MyProvider.MyCipher"));
    }

    private static final class ProviderService extends Provider.Service {

        ProviderService(Provider p, String type, String algo, String cn) {
            super(p, type, algo, cn, null, null);
        }

        ProviderService(Provider p, String type, String algo, String cn,
                String[] aliases, HashMap<String, String> attrs) {
            super(p, type, algo, cn,
                    (aliases == null ? null : Arrays.asList(aliases)), attrs);
        }

        @Override
        public Object newInstance(Object ctrParamObj)
                throws NoSuchAlgorithmException {

            String type = getType();
            if (ctrParamObj != null) {
                throw new InvalidParameterException(
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                        "constructorParameter not used with " + type
                        + " engines");
                }
                String algo = getAlgorithm();
                try {
                    if (type.equals("Cipher")) {
                        if (algo.equals("MyCipher")) {
                            return new MyCipher();
                        }
                    }
                } catch (Exception ex) {
                    throw new NoSuchAlgorithmException(
                            "Error constructing " + type + " for "
                            + algo + " using SunMSCAPI", ex);
                }
                throw new ProviderException("No impl for " + algo
                        + " " + type);
            }
        }

        @Override
        public String toString() {
            return "MyProvider [getName()=" + getName()
                    + ", getVersionStr()=" + getVersionStr() + ", getInfo()="
                    + getInfo() + "]";
        }
}
```

src/com.example.MyProvider/com/example/MyProvider/MyCipher.java

The `MyCipher` class extends the `CipherSPI`, which is a Server Provider Interface (SPI). Each cryptographic service that a provider implements has a subclass of the appropriate SPI. See Step 1: Write your Service Implementation Code.

> **Note**
>
> This code is only a stub provider that demonstrates how to write a provider; it's missing the actual cryptographic algorithm implementation. The `MyCipher` class would contain an actual cryptographic algorithm implementation if MyProvider were a real security provider.

```
package com.example.MyProvider;

import java.security.*;
import java.security.spec.*;
import javax.crypto.*;

/**
 * Implementation represents a test Cipher.
 *
 * All are stubs.
 */
public class MyCipher extends CipherSpi {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    @Override
    protected byte[] engineDoFinal(byte[] input, int inputOffset, int
inputLen)
            throws IllegalBlockSizeException, BadPaddingException {
        return null;
    }

    @Override
    protected int engineDoFinal(byte[] input, int inputOffset, int inputLen,
            byte[] output, int outputOffset) throws ShortBufferException,
            IllegalBlockSizeException, BadPaddingException {
        return 0;
    }

    @Override
    protected int engineGetBlockSize() {
        return 0;
    }

    @Override
    protected byte[] engineGetIV() {
        return null;
    }

    @Override
    protected int engineGetOutputSize(int inputLen) {
        return 0;
    }

    @Override
    protected AlgorithmParameters engineGetParameters() {
        return null;
    }

    @Override
    protected void engineInit(int opmode, Key key, SecureRandom random)
            throws InvalidKeyException {
    }

    @Override
    protected void engineInit(int opmode, Key key,
            AlgorithmParameterSpec params, SecureRandom random)
            throws InvalidKeyException, InvalidAlgorithmParameterException {
    }

    @Override
    protected void engineInit(int opmode, Key key, AlgorithmParameters params,
            SecureRandom random) throws InvalidKeyException,
            InvalidAlgorithmParameterException {
    }

    @Override
    protected void engineSetMode(String mode) throws NoSuchAlgorithmException
{
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    @Override
    protected void engineSetPadding(String padding)
            throws NoSuchPaddingException {
    }

    @Override
    protected int engineGetKeySize(Key key)
            throws InvalidKeyException {
        return 0;
    }

    @Override
    protected byte[] engineUpdate(byte[] input, int inputOffset, int
inputLen) {
        return null;
    }

    @Override
    protected int engineUpdate(byte[] input, int inputOffset, int inputLen,
            byte[] output, int outputOffset) throws ShortBufferException {
        return 0;
    }
}
```

src/com.example.MyProvider/META-INF/services/java.security.Provider

The `java.security.Provider` file enables automatic or unnamed modules to use the `ServiceLoader` class to search for your providers. See Step 6: Place Your Provider in a JAR File.

```
com.example.MyProvider.MyProvider
```

src/com.example.MyApp/module-info.java

This file contains a `uses` directive, which specifies a service that the module requires. This directive helps the module system locate providers and ensure that they run reliably. This is the complement to the `provides` directive in the `MyProvider` module definition.

```
module com.example.MyApp {
    uses java.security.Provider;
}
```

src/com.example.MyApp/com/example/MyApp/MyApp.java

```
package com.example.MyApp;

import java.util.*;
import java.security.*;
import javax.crypto.*;

/**
 * A simple JCE test client to access a simple test Provider/Cipher
 * implementation in a signed modular jar.
 */
public class MyApp {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    private static final String PROVIDER = "MyProvider";
    private static final String CIPHER = "MyCipher";

    public static void main(String[] args) throws Exception {

        /*
         * Registers MyProvider dynamically.
         *
         * Could do statically by editing the java.security file.
         * Use the first form if using ServiceLoader ("uses" or
         * META-INF/service), the second if using the traditional class
         * lookup method.  Both if provider could be deployed to either.
         *
         * security.provider.14=MyProvider
         * security.provider.15=com.example.MyProvider.MyProvider
         */
        ServiceLoader<Provider> sl =
            ServiceLoader.load(java.security.Provider.class);
        for (Provider p : sl) {
            if (p.getName().equals(PROVIDER)) {
                System.out.println("Registering the Provider");
                Security.addProvider(p);
            }
        }

        /*
         * Get a MyCipher from MyProvider and initialize it.
         */
        Cipher cipher = Cipher.getInstance(CIPHER, PROVIDER);
        cipher.init(Cipher.ENCRYPT_MODE, (Key) null);

        /*
         * What Provider did we get?
         */
        Provider p = cipher.getProvider();
        Class c = p.getClass();
        Module m = c.getModule();
        System.out.println(p.getName() + ": version "
            + p.getVersionStr() + "\n"
            + p.getInfo() + "\n    "
            + ((m.getName() == null) ? "<UNNAMED>" : m.getName())
            + "/" + c.getName());
    }
}
```

RunTest.sh

```
#!/bin/sh

#
# A simple example to show how a JCE provider could be developed in a
# modular JDK, for deployment as either Named/Unnamed modules.
#
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
#
# Edit as appropriate
#
JDK_DIR=d:/java/jdk9
KEYSTORE=YourKeyStore
STOREPASS=YourStorePass
SIGNER=YourAlias

echo "-----------"
echo "Clean/Init"
echo "-----------"
rm -rf mods jars
mkdir mods jars

echo "--------------------"
echo "Compiling MyProvider"
echo "--------------------"
${JDK_DIR}/bin/javac.exe \
    --module-source-path src \
    -d mods \
    $(find src/com.example.MyProvider -name '*.java' -print)

echo "-----------------------------------"
echo "Packaging com.example.MyProvider.jar"
echo "-----------------------------------"
${JDK_DIR}/bin/jar.exe --create \
    --file jars/com.example.MyProvider.jar \
    --verbose \
    --module-version=1.0 \
    -C mods/com.example.MyProvider . \
    -C src/com.example.MyProvider META-INF/services

echo "---------------------------------"
echo "Signing com.example.MyProvider.jar"
echo "---------------------------------"
${JDK_DIR}/bin/jarsigner.exe \
    -keystore ${KEYSTORE} \
    -storepass ${STOREPASS} \
    jars/com.example.MyProvider.jar ${SIGNER}

echo "---------------"
echo "Compiling MyApp"
echo "---------------"
${JDK_DIR}/bin/javac.exe \
    --module-source-path src \
    -d mods \
    $(find src/com.example.MyApp -name '*.java' -print)

echo "-----------------------------"
echo "Packaging com.example.MyApp.jar"
echo "-----------------------------"
${JDK_DIR}/bin/jar.exe --create \
    --file jars/com.example.MyApp.jar \
    --verbose \
    --module-version=1.0 \
    -C mods/com.example.MyApp .
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
echo "------------------------"
echo "Test1                   "
echo "Named Provider/Named App"
echo "------------------------"
${JDK_DIR}/bin/java.exe \
    --module-path 'jars' \
    -m com.example.MyApp/com.example.MyApp.MyApp

echo "--------------------------"
echo "Test2                     "
echo "Named Provider/Unnamed App"
echo "--------------------------"
${JDK_DIR}/bin/java.exe \
    --module-path 'jars/com.example.MyProvider.jar' \
    --class-path 'jars/com.example.MyApp.jar' \
    com.example.MyApp.MyApp

echo "--------------------------"
echo "Test3                     "
echo "Unnamed Provider/Named App"
echo "--------------------------"
${JDK_DIR}/bin/java.exe \
    --module-path 'jars/com.example.MyApp.jar' \
    --class-path 'jars/com.example.MyProvider.jar' \
    -m com.example.MyApp/com.example.MyApp.MyApp

echo "----------------------------"
echo "Test4                       "
echo "Unnamed Provider/Unnamed App"
echo "----------------------------"
${JDK_DIR}/bin/java.exe \
    --class-path \
        'jars/com.example.MyProvider.jar;jars/com.example.MyApp.jar' \
    com.example.MyApp.MyApp
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

4

JDK Providers Documentation

This document contains the technical details of the providers that are included in the JDK. It is assumed that readers have a strong understanding of the Java Cryptography Architecture and Provider Architecture.

> **Note**
>
> The Java Security Standard Algorithm Names contains more information about the standard names used in this document.

Topics

Introduction to JDK Providers

Import Limits on Cryptographic Algorithms

Cipher Transformations

SecureRandom Implementations

The SunPKCS11 Provider

The SUN Provider

The SunRsaSign Provider

The SunJSSE Provider

The SunJCE Provider

The SunJGSS Provider

The SunSASL Provider

The XMLDSig Provider

The SunPCSC Provider

The SunMSCAPI Provider

The SunEC Provider

The Apple Provider

The JdkLDAP Provider

The JdkSASL Provider

Introduction to JDK Providers

The Java platform defines a set of APIs spanning major security areas, including cryptography, public key infrastructure, authentication, and secure communication. These APIs enable developers to easily integrate security mechanisms into their application code.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Java Cryptography Architecture (JCA) and its Provider Architecture are core concepts of the Java Development Kit (JDK). It is assumed that readers have a solid understanding of this architecture.

Reminder: Cryptographic implementations in the JDK are distributed through several different providers ("SUN", "SunJSSE", "SunJCE", "SunRsaSign") for both historical reasons and by the types of services provided. General purpose applications **SHOULD NOT** request cryptographic services from specific providers. That is:

```
getInstance("...", "SunJCE");  // not recommended
```

versus

```
getInstance("...");            // recommended
```

Otherwise, applications are tied to specific providers that may not be available on other Java implementations. They also might not be able to take advantage of available optimized providers (for example, hardware accelerators via PKCS11 or native OS implementations such as Microsoft's MSCAPI) that have a higher preference order than the specific requested provider.

The following table lists the modules and the supported Java Cryptographic Service Providers:

Table 4-1 Modules and the Java Cryptographic Service Providers

Module	Provider(s)
java.base	SUN, SunRsaSign, SunJSSE, SunJCE, Apple
java.naming	JdkLDAP
java.security.jgss	SunJGSS
java.security.sasl	SunSASL
java.smartcardio	SunPCSC
java.xml.crypto	XMLDSig
jdk.crypto.cryptoki	SunPKCS11
jdk.crypto.ec	SunEC
jdk.crypto.mscapi	SunMSCAPI
jdk.security.jgss	JdkSASL

Import Limits on Cryptographic Algorithms

By default, an application can use cryptographic algorithms of any strength. However, due to import regulations in some locations, you may have to limit the strength of those algorithms. The JDK provides two different sets of jurisdiction policy files in the directory `<java-home>/conf/security/policy` that determine the strength of cryptographic algorithms. Information about jurisdiction policy files and how to activate them is available in Cryptographic Strength Configuration.

Consult your export/import control counsel or attorney to determine the exact requirements for your location.

Copyright © 1993, 2026, Oracle and/or its affiliates.

For the "limited" configuration, the following table lists the maximum key sizes allowed by the "limited" set of jurisdiction policy files:

Table 4-2 Maximum Keysize of Cryptographic Algorithms

Algorithm	Maximum Keysize
DES	64
DESede	*
RC2	128
RC4	128
RC5	128
RSA	*
all others	128

Cipher Transformations

The `javax.crypto.Cipher.getInstance(String transformation)` factory method generates `Cipher` objects using transformations of the form **algorithm/mode/padding**. If the mode/padding are omitted, the SunJCE and SunPKCS11 providers use ECB as the default mode and PKCS5Padding as the default padding for many symmetric ciphers.

It is recommended to use transformations that fully specify the algorithm, mode, and padding instead of relying on the defaults. The defaults are provider specific and can vary among providers.

> **Note**
>
> ECB mode is the easiest block cipher mode to use and is the default cipher mode. ECB works well for single blocks of data and can be parallelized but generally should not be used for encrypting multiple data blocks due to characteristics of the mode. This could result in trivial and full disclosure of confidential data. While this mode is available for use, it should only be used with an understanding of the cryptographic risks involved.

SecureRandom Implementations

The following table lists the default preference order of the available `SecureRandom` implementations.

Table 4-3 Default SecureRandom Implementations

OS	Algorithm Name	Provider Name
Linux	1. NativePRNG[1]	SUN
	2. DRBG	SUN
	3. SHA1PRNG [1]	SUN
	4. NativePRNGBlocking	SUN
	5. NativePRNGNonBlocking	SUN
macOS	1. NativePRNG[1]	SUN
	2. DRBG	SUN

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-3 (Cont.) Default SecureRandom Implementations

OS	Algorithm Name	Provider Name
	3. SHA1PRNG[1]	SUN
	4. NativePRNGBlocking	SUN
	5. NativePRNGNonBlocking	SUN
Windows	1. DRBG	SUN
	2. SHA1PRNG	SUN
	3. Windows-PRNG[2]	SunMSCAPI

1 On Linux and macOS, if the *entropy gathering device* in `java.security` is set to `file:/dev/urandom` or `file:/dev/random`, then NativePRNG is preferred to SHA1PRNG. Otherwise, SHA1PRNG is preferred.
2 There is currently no NativePRNG on Windows. Access to the equivalent functionality is via the SunMSCAPI provider.

The SunPKCS11 Provider

The Cryptographic Token Interface Standard (PKCS#11) provides native programming interfaces to cryptographic mechanisms, such as hardware cryptographic accelerators and Smart Cards. When properly configured, the `SunPKCS11` provider enables applications to use the standard JCA/JCE APIs to access native PKCS#11 libraries. **The `SunPKCS11` provider itself does not contain cryptographic functionality**, it is simply a conduit between the Java environment and the native PKCS11 providers. The PKCS#11 Reference Guide has a much more detailed treatment of this provider.

The SUN Provider

Algorithms

The following algorithms are available in the `SUN` provider:

Table 4-4 Algorithms in SUN provider

Engine	Algorithm Names
`AlgorithmParameterGenerator`	DSA
`AlgorithmParameters`	DSA
`CertificateFactory`	X.509
`CertPathBuilder`	PKIX
`CertPathValidator`	PKIX
`CertStore`	Collection
`Configuration`	JavaLoginConfig
`KeyFactory`	DSA HSS/LMS[1] ML-DSA ML-DSA-44 ML-DSA-65 ML-DSA-87

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-4 (Cont.) Algorithms in SUN provider

Engine	Algorithm Names
`KeyPairGenerator`	DSA[2] ML-DSA[3] ML-DSA-44 ML-DSA-65 ML-DSA-87
`KeyStore`	PKCS12[4] JKS[5] DKS CaseExactJKS
`MessageDigest`	MD2 MD5 SHA-1 SHA-224 SHA-256 SHA-384 SHA-512 SHA-512/224 SHA-512/256 SHA3-224 SHA3-256 SHA3-384 SHA3-512 SHAKE128-256 SHAKE256-512

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-4 (Cont.) Algorithms in SUN provider

Engine	Algorithm Names
`SecureRandom`	DRBG (The following mechanisms and algorithms are supported: Hash_DRBG and HMAC_DRBG with SHA-224, SHA-512/224, SHA-256, SHA-512/256, SHA-384 and SHA-512. CTR_DRBG (both use derivation function and not use) with AES-128, AES-192 and AES-256. Prediction resistance and reseeding supported for each combination, and security strength can be requested from 112 up to the highest strength one supports.) SHA1PRNG (Initial seeding is currently done via a combination of system attributes and the `java.security` entropy gathering device.) NativePRNG (`nextBytes()` uses `/dev/urandom`, `generateSeed()` uses `/dev/random`) NativePRNGBlocking (`nextBytes()` and `generateSeed()` use `/dev/random`) NativePRNGNonBlocking (`nextBytes()` and `generateSeed()` use `/dev/urandom`)

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-4 (Cont.) Algorithms in SUN provider

Engine	Algorithm Names
`Signature`	HSS/LMS[6]
	ML-DSA
	ML-DSA-44
	ML-DSA-65
	ML-DSA-87
	NONEwithDSA
	SHA1withDSA
	SHA224withDSA
	SHA256withDSA
	SHA384withDSA
	SHA512withDSA
	NONEwithDSAinP1363Format
	SHA1withDSAinP1363Format
	SHA224withDSAinP1363Format
	SHA256withDSAinP1363Format
	SHA384withDSAinP1363Format
	SHA512withDSAinP1363Format
	SHA3-224withDSA
	SHA3-256withDSA
	SHA3-384withDSA
	SHA3-512withDSA
	SHA3-224withDSAinP1363Format
	SHA3-256withDSAinP1363Format
	SHA3-384withDSAinP1363Format
	SHA3-512withDSAinP1363Format

Note

For signature generation, if the security strength of the digest algorithm is weaker than the security strength of the key used to sign the signature (for example, using (2048, 256)-bit DSA keys with the SHA1withDSA signature), then the operation will fail with the error message: "The security strength of SHA1 digest algorithm is not sufficient for this key size."

Copyright © 1993, 2026, Oracle and/or its affiliates.

[1] The `KeyFactory` implementation of HSS/LMS supports only public key management. An `InvalidKeySpecException` is thrown if the `generatePrivate` method is called or the `getKeySpec` and `translateKey` methods are called with a private key.

[2] If the `KeyPairGenerator` generates a key pair with the DSA algorithm, and the keysize (modulus size) is 512, 768, 1024, or 2048, then the `SUN` provider uses a set of precomputed values for the p, q, and g parameters. If the modulus size is not one of these values, then the `SUN` provider creates a new set of parameters.

[3] The ML-DSA `KeyPairGenerator` algorithm can be initialized with ML-DSA-44, ML-DSA-65, or ML-DSA-87 parameters. If you initialize the ML-DSA `KeyPairGenerator` algorithm without any parameters, then by default it uses ML-DSA-65 parameters. See `ParameterSpec` Names in Java Security Standard Algorithm Names.

[4] The PKCS12 `KeyStore` implementation does not support the `KeyBag` type. You can create a password-less PKCS12 keystore file by calling the `KeyStore.store()` method with a null password. The certificates in this keystore are stored unencrypted, and no MacData is added. This keystore can be loaded with any password (including null). Note that keys in a keystore are always stored encrypted.

[5] JKS and JCEKS keystores will be removed in a future release because they use outdated cryptographic algorithms. It is advised that you migrate to a PKCS12 keystore by using the `keytool -importkeystore` command.

[6] The `Signature` implementation of HSS/LMS only supports signature verification. An `InvalidKeyException` is thrown if the `initSign` method is called with a private key.

Object Identifiers Associated with SHA Message Digests and DSA Signatures

The following table lists object identifiers (OIDs) associated with SHA Message Digests:

Table 4-5 OIDs associated with SHA Message Digests

SHA Message Digest	OID
SHA-224	2.16.840.1.101.3.4.2.4
SHA-256	2.16.840.1.101.3.4.2.1
SHA-384	2.16.840.1.101.3.4.2.2
SHA-512	2.16.840.1.101.3.4.2.3
SHA-512/224	2.16.840.1.101.3.4.2.5
SHA-512/256	2.16.840.1.101.3.4.2.6
SHA3-224	2.16.840.1.101.3.4.2.7
SHA3-256	2.16.840.1.101.3.4.2.8
SHA3-384	2.16.840.1.101.3.4.2.9
SHA3-512	2.16.840.1.101.3.4.2.10

The following table lists OIDs associated with DSA Signatures:

Table 4-6 OIDs associated with DSA Signatures

DSA Signature	OID
ML-DSA-44	2.16.840.1.101.3.4.3.17
ML-DSA-65	2.16.840.1.101.3.4.3.18
ML-DSA-87	2.16.840.1.101.3.4.3.19
SHA1withDSA	1.2.840.10040.4.3 1.3.14.3.2.13 1.3.14.3.2.27
SHA224withDSA	2.16.840.1.101.3.4.3.1
SHA256withDSA	2.16.840.1.101.3.4.3.2
SHA384withDSA	2.16.840.1.101.3.4.3.3
SHA512withDSA	2.16.840.1.101.3.4.3.4

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-6 (Cont.) OIDs associated with DSA Signatures

DSA Signature	OID
SHA3-224withDSA	2.16.840.1.101.3.4.3.5
SHA3-256withDSA	2.16.840.1.101.3.4.3.6
SHA3-384withDSA	2.16.840.1.101.3.4.3.7
SHA3-512withDSA	2.16.840.1.101.3.4.3.8

Keysize Restrictions

The `SUN` provider uses the following default keysizes (in bits) and enforces the following restrictions:

Table 4-7 KeyPairGenerator Algorithm Keysize Restrictions

Algorithm Name	Default Keysize	Restrictions/Comments
DSA	2048	Keysize must be a multiple of 64, ranging from 512 to 1024, plus 2048 and 3072.

Table 4-8 AlgorithmParameterGenerator Algorithm Keysize Restrictions

Algorithm Name	Default Keysize	Restrictions/Comments
DSA	2048	Keysize must be a multiple of 64, ranging from 512 to 1024, plus 2048 and 3072.

CertificateFactory/CertPathBuilder/CertPathValidator/CertStore Implementations

See Appendix B: CertPath Implementation in SUN Provider in the Java PKI Programmer's Guide for details of the `SUN` provider implementations for `CertificateFactory`, `CertPathBuilder`, `CertPathValidator`, and `CertStore`.

The SunRsaSign Provider

Algorithms

The following algorithms are available in the `SunRsaSign` provider:

Table 4-9 SunRsaSign Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`AlgorithmParameters`	RSASSA-PSS
`KeyFactory`	RSA RSASSA-PSS
`KeyPairGenerator`	RSA RSASSA-PSS

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-9 (Cont.) SunRsaSign Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`Signature`	MD2withRSA
	MD5withRSA
	SHA1withRSA
	SHA224withRSA
	SHA256withRSA
	SHA384withRSA
	SHA512withRSA
	SHA512/224withRSA
	SHA512/256withRSA
	SHA3-224withRSA
	SHA3-256withRSA
	SHA3-384withRSA
	SHA3-512withRSA
	RSASSA-PSS[1]

[1] The RSASSA-PSS signature implementation supports the MGF1 mask generation function and the SHA-1, SHA-2, and SHA-3 families of digest algorithms. See RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2, Appendix A.2.3, "RSASSA-PSS".

Keysize Restrictions

The `SunRsaSign` provider uses the following default keysize (in bits) and enforces the following restriction:

Table 4-10 SunRsaSign Provider Keysize Restrictions

Alg. Name	Default Keysize	Restrictions/Comments
RSA and RSASSA-PSS	3072	Keysize must range between 512 and 16384 bits. If the key size exceeds 3072, then the public exponent length cannot exceed 64 bits.

The SunJSSE Provider

Algorithms

The following algorithms are available in the `SunJSSE` provider:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-11 Algorithms in SunJSSE Provider

Engine	Algorithm Name(s)
`KeyManagerFactory`	PKIX: A factory for `X509ExtendedKeyManager` instances that manage X.509 certificate-based key pairs for local side authentication according to the rules defined by the IETF PKIX working group in RFC 5280. This `KeyManagerFactory` currently supports initialization using a `KeyStore` object or `javax.net.ssl.KeyStoreBuilderParameters`. SunX509: A `KeyManagerFactory` that is the same as PKIX but with better performance at the expense of not supporting initialization with the class `javax.net.ssl.ManagerFactoryParameters`. SunX509 supports initialization using a `Keystore` object only. SunX509 caches its whole key store during initialization thus improving performance. This means that subsequent `KeyStore` modifications have no effect on SunX509, unlike PKIX. Restore the legacy behavior of SunX509, which is to not check certificates, by setting the `jdk.tls.SunX509KeyManager.certChecking` system property to `false`.
`KeyStore`	PKCS12 **Note:** The SunJSSE provider is for backwards compatibility with older releases, and should no longer be used for `KeyStore`.
`SSLContext`	SSL SSLv3 TLS TLSv1 TLSv1.1 TLSv1.2 TLSv1.3 DTLS DTLSv1.0 DTLSv1.2

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-11 (Cont.) Algorithms in SunJSSE Provider

Engine	Algorithm Name(s)
`TrustManagerFactory`	PKIX: A factory for `X509ExtendedTrustManager` instances that validate certificate chains according to the rules defined by the IETF PKIX working group in RFC 5280. This `TrustManagerFactory` currently supports initialization using a `KeyStore` object or `javax.net.ssl.CertPathTrustManagerParameters`. SunX509: A factory for `X509ExtendedTrustManager` instances that validate certificate chains, but with less strict checking of certificate usage/validity and chain verification. This `TrustManagerFactory` supports initialization using a `Keystore` object, but does not currently support initialization using the class `javax.net.ssl.ManagerFactoryParameters`. **Note:** The SunX509 factory is for backwards compatibility with older releases, and should no longer be used.

SunJSSE Provider Protocol Parameters

The `SunJSSE` provider supports the `protocol` parameters listed in Table 4-12.

Table 4-12 SunJSSE Provider Protocol Versions

Protocol Version	Enabled by Default for Client	Enabled by Default for Server
SSLv3	No	No
TLSv1[1]	Yes	Yes
TLSv1.1[1]	Yes	Yes
TLSv1.2	Yes	Yes
TLSv1.3	Yes	Yes
SSLv2Hello	No	No
DTLSv1.0	Yes	Yes
DTLSv1.2	Yes	Yes

[1] TLS 1.0 and 1.1 are versions of the TLS protocol that are no longer considered secure and have been superseded by more secure and modern versions (TLS 1.2 and 1.3). These versions have now been disabled by default. If you encounter issues, you can, at your own risk, re-enable the versions by removing `TLSv1` or `TLSv1.1` from the `jdk.tls.disabledAlgorithms` Security Property in the `java.security` configuration file.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The protocols available by default in a JDK release change as new protocols are developed and old protocols are found to be less effective than previously thought. The JDK uses two mechanisms to restrict the availability of these protocols:
>
> - The `jdk.tls.disabledAlgorithms` Security Property: This disables categories of protocols and cipher suites. For example, if this Security Property contains `SSLv3`, then the SSLv3 protocol would be disabled. See Disabled and Restricted Cryptographic Algorithms for information about this Security Property.
> - Moving the protocol to the list of protocols not enabled by default as indicated in Table 4-12.

The enabled protocol versions of an `SSLContext` implementation may differ from the default values in the previous table depending on the algorithm and its mode (client or server). The following tables list the enabled protocol versions for `SSLContext` implementations that differ from the default:

Table 4-13 Enabled Protocol Versions for Specific SSLContext Implementations in Client Mode

SSLContext Algorithm	SSL/TLS/DTLS Protocol Version							
	SSLv2Hello	SSLv3	TLSv1	TLSv1.1	TLSv1.2	TLSv1.3	DTLSv1.0	DTLSv1.2
SSLv3	No	No	Yes	No	No	No	N/A	N/A
TLSv1	No	No	Yes	No	No	No	N/A	N/A
TLSv1.1	No	No	Yes	Yes	No	No	N/A	N/A
TLSv1.2	No	No	Yes	Yes	Yes	No	N/A	N/A
TLSv1.3	No	No	Yes	Yes	Yes	Yes	N/A	N/A
Default	No	No	Yes	Yes	Yes	Yes	N/A	N/A
TLS	No	No	Yes	Yes	Yes	Yes	N/A	N/A
SSL	No	No	Yes	Yes	Yes	Yes	N/A	N/A
DTLSv1.0	N/A	N/A	N/A	N/A	N/A	N/A	Yes	No
DTLSv1.2	N/A	N/A	N/A	N/A	N/A	N/A	Yes	Yes
DTLS	N/A	N/A	N/A	N/A	N/A	N/A	Yes	Yes

Table 4-14 Enabled Protocol Versions for Specific SSLContext Implementations in Server Mode

SSLContext Algorithm	SSL/TLS/DTLS Protocol Version							
	SSLv2Hello	SSLv3	TLSv1	TLSv1.1	TLSv1.2	TLSv1.3	DTLSv1.0	DTLSv1.2
SSLv3	No	No	Yes	Yes	Yes	Yes	N/A	N/A
TLSv1	No	No	Yes	Yes	Yes	Yes	N/A	N/A
TLSv1.1	No	No	Yes	Yes	Yes	Yes	N/A	N/A
TLSv1.2	No	No	Yes	Yes	Yes	Yes	N/A	N/A
TLSv1.3	No	No	Yes	Yes	Yes	Yes	N/A	N/A
Default	No	No	Yes	Yes	Yes	Yes	N/A	N/A

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-14 (Cont.) Enabled Protocol Versions for Specific SSLContext Implementations in Server Mode

SSLContext Algorithm	SSL/TLS/DTLS Protocol Version							
	SSLv2Hello	SSLv3	TLSv1	TLSv1.1	TLSv1.2	TLSv1.3	DTLSv1.0	DTLSv1.2
TLS	No	No	Yes	Yes	Yes	Yes	N/A	N/A
SSL	No	No	Yes	Yes	Yes	Yes	N/A	N/A
DTLSv1.0	N/A	N/A	N/A	N/A	N/A	N/A	Yes	Yes
DTLSv1.2	N/A	N/A	N/A	N/A	N/A	N/A	Yes	Yes
DTLS	N/A	N/A	N/A	N/A	N/A	N/A	Yes	Yes

SunJSSE Cipher Suites

The following are the currently implemented SunJSSE cipher suites for this JDK release, sorted by order of preference. Not all of these cipher suites are available for use by default. See JSSE Cipher Suite Names in *Java Security Standard Algorithm Names* to determine which protocols that each cipher suite supports.

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_DSS_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_RSA_WITH_AES_256_CBC_SHA256

Copyright © 1993, 2026, Oracle and/or its affiliates.

- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
- TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_DSS_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- TLS_EMPTY_RENEGOTIATION_INFO_SCSV
- TLS_DH_anon_WITH_AES_256_GCM_SHA384
- TLS_DH_anon_WITH_AES_128_GCM_SHA256
- TLS_DH_anon_WITH_AES_256_CBC_SHA256
- TLS_ECDH_anon_WITH_AES_256_CBC_SHA
- TLS_DH_anon_WITH_AES_256_CBC_SHA

Copyright © 1993, 2026, Oracle and/or its affiliates.

- TLS_DH_anon_WITH_AES_128_CBC_SHA256
- TLS_ECDH_anon_WITH_AES_128_CBC_SHA
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_SHA
- TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- TLS_ECDH_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_MD5
- TLS_ECDH_anon_WITH_RC4_128_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DHE_RSA_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- SSL_DH_anon_WITH_DES_CBC_SHA
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- TLS_RSA_WITH_NULL_SHA256
- TLS_ECDHE_ECDSA_WITH_NULL_SHA
- TLS_ECDHE_RSA_WITH_NULL_SHA
- SSL_RSA_WITH_NULL_SHA
- TLS_ECDH_ECDSA_WITH_NULL_SHA
- TLS_ECDH_RSA_WITH_NULL_SHA
- TLS_ECDH_anon_WITH_NULL_SHA
- SSL_RSA_WITH_NULL_MD5

Note

- The cipher suite order of preference may change in future releases.
- TLS_EMPTY_RENEGOTIATION_INFO_SCSV is a pseudo-cipher suite that supports RFC 5746.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The cipher suites available by default in a JDK release change as new algorithms are developed and old algorithms are found to be less effective than previously thought. Oracle JDK uses two mechanisms to restrict the availability of these algorithms:

- The `jdk.tls.disabledAlgorithms` Security Property, which disables categories of cipher suites. For example, if this Security Property contains `RC4`, then all RC4-based cipher suites would be disabled.
- Moving the cipher suite to the list of suites not enabled by default.

See Disabled and Restricted Cryptographic Algorithms for information about the `jdk.tls.disabledAlgorithms` Security Property.

Determining Current List of Protocols and Cipher Suites Available by Default

To obtain the current list of protocols and cipher suites that are available by default, run one of the following commands:

```
keytool -showinfo -tls
java -XshowSettings:security:tls
```

Note that the list generated by these commands don't include suites that the `jdk.tls.disabledAlgorithms` Security Property disabled.

Tighter Checking of EncryptedPreMasterSecret Version Numbers

Prior to the JDK 7 release, the SSL/TLS implementation did not check the version number in PreMasterSecret, and the SSL/TLS client did not send the correct version number by default. Unless the system property `com.sun.net.ssl.rsaPreMasterSecretFix` is set to `true`, the TLS client sends the active negotiated version, but not the expected maximum version supported by the client.

For compatibility, this behavior is preserved for SSL version 3.0 and TLS version 1.0. However, for TLS version 1.1 or later, the implementation tightens checking the PreMasterSecret version numbers as required by RFC 5246. Clients always send the correct version number, and servers check the version number strictly. The system property, `com.sun.net.ssl.rsaPreMasterSecretFix`, is not used in TLS 1.1 or later.

Named Groups

Named groups are predefined cryptographic parameters for key exchange. The standard list of key exchange named groups are defined in the Named Groups section of *Java Security Standard Algorithm Names Specification*. The following table lists the named groups implemented and supported by the SunJSSE provider:

Table 4-15 Named Groups Implemented and Supported by the SunJSSE Provider

Named Group	Enabled by Default?	Description
ffdhe2048 ffdhe3072 ffdhe4096	Yes	The Finite Field Diffie-Hellman Ephemeral (FFDHE) groups as specified in RFC 7919.
ffdhe6144 ffdhe8192	No	The Finite Field Diffie-Hellman Ephemeral (FFDHE) groups as specified in RFC 7919.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-15 (Cont.) Named Groups Implemented and Supported by the SunJSSE Provider

Named Group	Enabled by Default?	Description
secp256r1 secp384r1 secp521r1	Yes	The NIST elliptic curves as specified in RFC 8422.
SecP256r1MLKEM768 SecP384r1MLKEM1024	No	The post-quantum hybrid key exchange schemes as defined in RFC 10024. See Enabling Hybrid Key Exchange Schemes for more information.
x25519 x448	Yes	The elliptic curves as specified in RFC 8446 and RFC 8422.
X25519MLKEM768	Yes	The post-quantum hybrid key exchange scheme as defined in RFC 10024. See Enabling Hybrid Key Exchange Schemes for more information.

See Customizing the Supported Named Groups for TLS/DTLS Key Exchange to customize which named groups are enabled and their preference order.

The SunJCE Provider

As described briefly in The SUN Provider, US export regulations at the time restricted the type of cryptographic functionality that could be available in the JDK. A separate API and reference implementation was developed that allowed applications to encrypt/decrypt date. The Java Cryptographic Extension (JCE) was released as a separate ”Optional Package” (also briefly known as a “Standard Extension”), and was available for JDK 1.2x and 1.3x. During the development of JDK 1.4, regulations were relaxed enough that JCE (and SunJSSE) could be bundled as part of the JDK.

The following algorithms are available in the SunJCE provider:

Table 4-16 The SunJCE Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`AlgorithmParameterGenerator`	DiffieHellman

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-16 (Cont.) The SunJCE Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`AlgorithmParameters`	AES Blowfish ChaCha20-Poly1305 DES DESede DiffieHellman GCM OAEP PBE PBES2 PBEWithHmacSHA1AndAES_128 PBEWithHmacSHA224AndAES_128 PBEWithHmacSHA256AndAES_128 PBEWithHmacSHA384AndAES_128 PBEWithHmacSHA512AndAES_128 PBEWithHmacSHA512/224AndAES_128 PBEWithHmacSHA512/256AndAES_128 PBEWithHmacSHA1AndAES_256 PBEWithHmacSHA224AndAES_256 PBEWithHmacSHA256AndAES_256 PBEWithHmacSHA384AndAES_256 PBEWithHmacSHA512/224AndAES_256 PBEWithHmacSHA512/256AndAES_256 PBEWithHmacSHA512AndAES_256 PBEWithMD5AndDES PBEWithMD5AndTripleDES PBEWithSHA1AndDESede PBEWithSHA1AndRC2_40 PBEWithSHA1AndRC2_128 PBEWithSHA1AndRC4_40 PBEWithSHA1AndRC4_128 RC2
`Cipher`	See Table 4-17
`KDF`	HKDF-SHA256 HKDF-SHA384 HKDF-SHA512
`KEM`	ML-KEM ML-KEM-512 ML-KEM-768 ML-KEM-1024 DHKEM[1]
`KeyAgreement`	DiffieHellman

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-16 (Cont.) The SunJCE Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`KeyFactory`	DiffieHellman ML-KEM ML-KEM-512 ML-KEM-768 ML-KEM-1024
`KeyGenerator`	AES ARCFOUR Blowfish ChaCha20 DES DESede HmacMD5 HmacSHA1 HmacSHA224 HmacSHA256 HmacSHA384 HmacSHA512 HmacSHA512/224 HmacSHA512/256 HmacSHA3-224 HmacSHA3-256 HmacSHA3-384 HmacSHA3-512 RC2
`KeyPairGenerator`	DiffieHellman ML-KEM [2] ML-KEM-512 ML-KEM-768 ML-KEM-1024
`KeyStore`	JCEKS[3]

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-16 (Cont.) The SunJCE Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
Mac	HmacMD5
	HmacSHA1
	HmacSHA224
	HmacSHA256
	HmacSHA384
	HmacSHA512
	HmacSHA512/224
	HmacSHA512/256
	HmacSHA3-224
	HmacSHA3-256
	HmacSHA3-384
	HmacSHA3-512
	HmacPBESHA1
	HmacPBESHA224
	HmacPBESHA256
	HmacPBESHA384
	HmacPBESHA512
	HmacPBESHA512/224
	HmacPBESHA512/256
	PBEWithHmacSHA1
	PBEWithHmacSHA224
	PBEWithHmacSHA256
	PBEWithHmacSHA384
	PBEWithHmacSHA512
	PBEWithHmacSHA512/224
	PBEWithHmacSHA512/256

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-16 (Cont.) The SunJCE Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`SecretKeyFactory`	DES
	DESede
	PBEWithMD5AndDES
	PBEWithMD5AndTripleDES
	PBEWithSHA1AndDESede
	PBEWithSHA1AndRC2_40
	PBEWithSHA1AndRC2_128
	PBEWithSHA1AndRC4_40
	PBEWithSHA1AndRC4_128
	PBEWithHmacSHA1AndAES_128
	PBEWithHmacSHA224AndAES_128
	PBEWithHmacSHA256AndAES_128
	PBEWithHmacSHA384AndAES_128
	PBEWithHmacSHA512AndAES_128
	PBEWithHmacSHA512/224AndAES_128
	PBEWithHmacSHA512/256AndAES_128
	PBEWithHmacSHA1AndAES_256
	PBEWithHmacSHA224AndAES_256
	PBEWithHmacSHA256AndAES_256
	PBEWithHmacSHA384AndAES_256
	PBEWithHmacSHA512/224AndAES_256
	PBEWithHmacSHA512/256AndAES_256
	PBEWithHmacSHA512AndAES_256
	PBKDF2WithHmacSHA1
	PBKDF2WithHmacSHA224
	PBKDF2WithHmacSHA256
	PBKDF2WithHmacSHA384
	PBKDF2WithHmacSHA512
	PBKDF2WithHmacSHA512/224
	PBKDF2WithHmacSHA512/256

[1] The DHKEM algorithm accepts EC keys on the secp256r1, secp384r1, and secp521r1 curves and XDH keys on Curve25519 and Curve448.

[2] The ML-KEM `KeyPairGenerator` algorithm can be initialized with ML-KEM-512, ML-KEM-768, or ML-KEM-1024 parameters. If you initialize the ML-KEM `KeyPairGenerator` algorithm without any parameters, then by default it uses ML-KEM-768 parameters. See `ParameterSpec` Names in Java Security Standard Algorithm Names

[3] JKS and JCEKS keystores will be removed in a future release because they use outdated cryptographic algorithms. It is advised that you migrate to a PKCS12 keystore by using the `keytool -importkeystore` command.

The following table lists cipher transformations available in the SunJCE provider.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-17 The SunJCE Provider Cipher Transformations

Algorithm Names	Modes	Paddings
AES	ECB, CBC, PCBC, CFB[1], CFB8..CFB128, OFB[1], OFB8..OFB128	NoPadding, PKCS5Padding, ISO10126Padding[2]
AES	CTR, CTS, GCM	NoPadding
AES_128, AES_192, AES_256	ECB, CBC, OFB, CFB, GCM	NoPadding
AESWrap	ECB	NoPadding
AESWrap_128	ECB	NoPadding
AESWrap_192	ECB	NoPadding
AESWrap_256	ECB	NoPadding
ARCFOUR	ECB	NoPadding
Blowfish, DES, DESede, RC2	ECB, CBC, PCBC, CTR, CTS, CFB[1], CFB8..CFB64, OFB[1], OFB8..OFB64	NoPadding, PKCS5Padding, ISO10126Padding
ChaCha20	None	NoPadding
ChaCha20-Poly1305	None	NoPadding
DESedeWrap	CBC	NoPadding
HPKE[3]	*Not applicable*	*Not applicable*
PBEWithHmacSHA1AndAES_128, PBEWithHmacSHA224AndAES_128, PBEWithHmacSHA256AndAES_128, PBEWithHmacSHA384AndAES_128, PBEWithHmacSHA512AndAES_128, PBEWithHmacSHA512/224AndAES_128, PBEWithHmacSHA512/256AndAES_128, PBEWithHmacSHA1AndAES_256, PBEWithHmacSHA224AndAES_256, PBEWithHmacSHA256AndAES_256, PBEWithHmacSHA384AndAES_256, PBEWithHmacSHA512/224AndAES_256, PBEWithHmacSHA512/256AndAES_256, PBEWithHmacSHA512AndAES_256, PBEWithMD5AndDES, PBEWithMD5AndTripleDES[4], PBEWithSHA1AndDESede, PBEWithSHA1AndRC2_40, PBEWithSHA1AndRC2_128, PBEWithSHA1AndRC4_40, PBEWithSHA1AndRC4_128	CBC	PKCS5Padding
RSA	ECB	NoPadding, PKCS1Padding, OAEPPadding, OAEPWithMD5AndMGF1Padding, OAEPWithSHA-1AndMGF1Padding, OAEPWithSHA-1AndMGF1Padding, OAEPWithSHA-224AndMGF1Padding, OAEPWithSHA-256AndMGF1Padding, OAEPWithSHA-384AndMGF1Padding, OAEPWithSHA-512AndMGF1Padding, OAEPWithSHA-512/224AndMGF1Padding, OAEPWithSHA-512/256AndMGF1Padding

Copyright © 1993, 2026, Oracle and/or its affiliates.

1 CFB/OFB with no specified value defaults to the block size of the algorithm. (i.e. AES is 128; Blowfish, DES, DESede, and RC2 are 64.)

2 Though the standard doesn't specify or require the padding bytes to be random, the Java SE ISO10126Padding implementation pads with random bytes (until the last byte, which provides the length of padding, as specified).

3 HPKE is not expressed as a transformation string in the form algorithm/mode/padding.

4 PBEWithMD5AndTripleDES is a proprietary algorithm that has not been standardized.

Keysize Restrictions

The SunJCE provider uses the following default key sizes (in bits) and enforces the following restrictions:

KeyGenerator

Table 4-18 The SunJCE Provider Key Size Restrictions

Algorithm Name	Default Key size	Restrictions/Comments
AES	256 if permitted by the cryptographic policy (see Import Limits on Cryptographic Algorithms), 128 otherwise	Key size must be equal to 128, 192, or 256.
ARCFOUR (RC4)	128	Key size must range between 40 and 1024 (inclusive).
Blowfish	128	Key size must be a multiple of 8, ranging from 32 to 448 (inclusive).
ChaCha20	256	Key size must be equal to 256.
DES	56	Key size must be equal to 56.
DESede (Triple DES)	168	Key size must be equal to 112 or 168. A key size of 112 will generate a Triple DES key with 2 intermediate keys, and a key size of 168 will generate a Triple DES key with 3 intermediate keys. Due to the "Meet-In-The-Middle" problem, even though 112 or 168 bits of key material are used, the **effective** key size is 80 or 112 bits respectively.
HmacMD5	512	No key size restriction.
HmacSHA1	512	No key size restriction.
HmacSHA224	224	No key size restriction.
HmacSHA256	256	No key size restriction.
HmacSHA384	384	No key size restriction.
HmacSHA512	512	No key size restriction.
RC2	128	Key size must range between 40 and 1024 (inclusive).

Note

The various Password-Based Encryption (PBE) algorithms use various algorithms to generate key data, and ultimately depends on the targeted Cipher algorithm. For example, PBEWithMD5AndDES will always generate 56–bit keys.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-19 KeyPairGenerator

Algorithm Name	Default Key size	Restrictions/Comments
Diffie-Hellman (DH)	3072	Key size must be a multiple of 64, ranging from 512 to 1024, plus 1536, 2048, 3072, 4096, 6144, 8192.

Table 4-20 AlgorithmParameterGenerator

Algorithm Name	Default Key size	Restrictions/Comments
Diffie-Hellman (DH)	3072	Key size must be a multiple of 64, ranging from 512 to 1024, plus 2048 and 3072.

The SunJGSS Provider

Algorithms

The following algorithms are available in the SunJGSS provider:

Table 4-21 SunJGSS Provider Algorithm Names

OID	Name
`1.2.840.113554.1.2.2`	Kerberos v5
`1.3.6.1.5.5.2`	SPNEGO

The SunSASL Provider

Algorithms

The following algorithms are available in the `SunSASL` provider:

Table 4-22 SunSASL Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`SaslClient`	CRAM-MD5 DIGEST-MD5 EXTERNAL NTLM PLAIN
`SaslServer`	CRAM-MD5 DIGEST-MD5 NTLM

Copyright © 1993, 2026, Oracle and/or its affiliates.

The XMLDSig Provider

Algorithms

The following algorithms are available in the `XMLDSig` provider:

Table 4-23 XMLDSig Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`KeyInfoFactory`	DOM
`TransformService`	• http://www.w3.org/TR/2001/REC-xml-c14n-20010315 – `CanonicalizationMethod.INCLUSIVE` • http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments – `CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS` • http://www.w3.org/2001/10/xml-exc-c14n# – `CanonicalizationMethod.EXCLUSIVE` • http://www.w3.org/2001/10/xml-exc-c14n#WithComments – `CanonicalizationMethod.EXCLUSIVE_WITH_COMMENTS` • http://www.w3.org/2000/09/xmldsig#base64 – `Transform.BASE64` • http://www.w3.org/2000/09/xmldsig#enveloped-signature – `Transform.ENVELOPED` • http://www.w3.org/TR/1999/REC-xpath-19991116 – `Transform.XPATH` • http://www.w3.org/2002/06/xmldsig-filter2 – `Transform.XPATH2` • http://www.w3.org/TR/1999/REC-xslt-19991116 – `Transform.XSLT`
`XMLSignatureFactory`	DOM

The SunPCSC Provider

The SunPCSC provider enables applications to use the Java Smart Card I/O API to interact with the PC/SC Smart Card stack of the underlying operating system. Consult your operating system documentation for details.

On Linux, SunPCSC accesses the PC/SC stack via the `libpcsclite.so` library. It looks for this library in the directories `/usr/$LIBISA` and `/usr/local/$LIBISA`, where `$LIBISA` is expanded to `lib64` on 64-bit Linux. The system property `sun.security.smartcardio.library` may also be set to the full filename of an alternate `libpcsclite.so` implementation. On Windows, SunPCSC always calls into `winscard.dll` and no Java-level configuration is necessary or possible.

If PC/SC is available on the host platform, the SunPCSC implementation can be obtained via `TerminalFactory.getDefault()` and `TerminalFactory.getInstance("PC/SC")`. If PC/SC is not available or not correctly configured, a `getInstance()` call will fail with a `NoSuchAlgorithmException` and `getDefault()` will return a JDK built-in implementation that does not support any terminals.

Algorithms

The following algorithms are available in the `SunPCSC` provider:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-24 The SunPCSC Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`TerminalFactory`	PC/SC

The SunMSCAPI Provider

The `SunMSCAPI` provider enables applications to use the standard JCA/JCE APIs to access the native cryptographic libraries, certificates stores and key containers on Windows. The `SunMSCAPI` provider itself does not contain cryptographic functionality, it is simply a conduit between the Java environment and the native cryptographic services on Windows.

Algorithms

The following algorithms are available in the `SunMSCAPI` provider:

Table 4-25 The SunMSCAPI Algorithm Names for Engine Classes

Engine	Algorithm Names
`Cipher`	RSA *RSA/ECB/PKCS1Padding only*
`KeyPairGenerator`	RSA
`KeyStore`	**Windows-MY-CURRENTUSER** (also known as **Windows-MY**): The keystore type that identifies the native Microsoft Windows MY keystore. It contains the user's personal certificates and associated private keys that are only accessible to the current user account. **Windows-ROOT-CURRENTUSER** (also known as **Windows-ROOT**): The keystore type that identifies the native Microsoft Windows ROOT keystore. It contains the certificates of Root certificate authorities and other self-signed trusted certificates that are only accessible to the current user account. **Windows-MY-LOCALMACHINE**: The keystore type that identifies the native Microsoft Windows MY keystore. It contains certificates and associated private keys that are accessible to all accounts on the system. **Windows-ROOT-LOCALMACHINE**: The keystore type that identifies the native Microsoft Windows ROOT keystore. It contains the certificates of Root certificate authorities and other self-signed trusted certificates that are accessible to all accounts on the system.
`SecureRandom`	**Windows-PRNG** : The name of the native pseudo-random number generation (PRNG) algorithm.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-25 (Cont.) The SunMSCAPI Algorithm Names for Engine Classes

Engine	Algorithm Names
`Signature`	MD5withRSA
	MD2withRSA
	NONEwithRSA
	SHA1withRSA
	SHA256withRSA
	SHA384withRSA
	SHA512withRSA
	RSASSA-PSS
	SHA1withECDSA
	SHA224withECDSA
	SHA256withECDSA
	SHA384withECDSA
	SHA512withECDSA

Keysize Restrictions

The SunMSCAPI provider uses the following default keysizes (in bits) and enforce the following restrictions:

KeyGenerator

Table 4-26 The SunMSCAPI Provider Keysize Restrictions

Alg. Name	Default Keysize	Restrictions/Comments
RSA	2048	Keysize ranges from 512 bits to 16,384 bits (depending on the underlying Microsoft Windows cryptographic service provider).

The SunEC Provider

The `SunEC` provider implements Elliptical Curve Cryptography (ECC). Compared to traditional cryptosystems such as RSA, ECC offers equivalent security with smaller key sizes, which results in faster computations, lower power consumption, and memory and bandwidth savings. Applications can use the standard JCA/JCE APIs to access ECC functionality without the dependency on external ECC libraries (through `SunPKCS11`).

Algorithms

The following algorithms are available in the `SunEC` provider:

Table 4-27 The SunEC Provider Names for Engine Classes

Engine	Algorithm Name(s)
`AlgorithmParameters`	EC

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-27 (Cont.) The SunEC Provider Names for Engine Classes

Engine	Algorithm Name(s)
`KeyAgreement`	ECDH X25519 X448 XDH
`KeyFactory`	EC Ed25519 Ed448 EdDSA X25519 X448 XDH
`KeyPairGenerator`	EC Ed25519 Ed448 EdDSA[1] X25519 X448 XDH [2]
`Signature`	Ed25519 Ed448 EdDSA NONEwithECDSA SHA1withECDSA SHA224withECDSA SHA256withECDSA SHA384withECDSA SHA512withECDSA NONEwithECDSAinP1363Format SHA1withECDSAinP1363Format SHA224withECDSAinP1363Format SHA256withECDSAinP1363Format SHA384withECDSAinP1363Format SHA512withECDSAinP1363Format SHA3-224withECDSA SHA3-256withECDSA SHA3-384withECDSA SHA3-512withECDSA SHA3-224withECDSAinP1363Format SHA3-256withECDSAinP1363Format SHA3-384withECDSAinP1363Format SHA3-512withECDSAinP1363Format

[1] The EdDSA `KeyPairGenerator` algorithm can be initialized with either Ed25519 or Ed448 parameters. If you initialize the EdDSA `KeyPairGenerator` algorithm without any parameters, then by default it uses

Copyright © 1993, 2026, Oracle and/or its affiliates.

Ed25519 parameters. All EdDSA variants — pure, prehashed, and context — are supported. See `ParameterSpec` Names in Java Security Standard Algorithm Names.

[2] The XDH `KeyPairGenerator` algorithm can be initialized with either X25519 or X448 parameters. If you initialize the XDH `KeyPairGenerator` algorithm without any parameters, then by default it uses X25519 parameters. See `ParameterSpec` Names in Java Security Standard Algorithm Names.

Keysize Restrictions

The SunEC provider uses the following default keysizes (in bits) and enforces the following restrictions:

Table 4-28 The SunEC Provider Keysize Restrictions

KeyPairGenerator Algorithm Name	Default Keysize	Restrictions/Comments
EC	384	Keysize must be 256, 384, or 521
Ed25519	255	Keysize must be 255
Ed448	448	Keysize must be 448
EdDSA	255	Keysize must be 255 or 448
X25519	255	Keysize must be 255
X448	448	Keysize must be 448
XDH	255	Keysize must be 255 or 448

Supported Elliptic Curve Names

The `SunEC` provider includes implementations of various elliptic curves for use with the EC, Elliptic-Curve Diffie-Hellman (ECDH), and Elliptic Curve Digital Signature Algorithm (ECDSA) algorithms. Some of these curves have been implemented using modern formulas and techniques that are valuable for preventing side-channel attacks. The others are legacy curves that might be more vulnerable to attacks and should not be used. The following tables list the curves that fall into each of these categories.

In the following tables, the first column, Curve Name, lists the name that `SunEC` implements. The second column, Object Identifier, specifies the EC name's object identifier. The third column, Additional Names/Aliases, specifies any additional names or aliases for that curve. (A value of N/A means that there are no additional names.) All strings that appear in one row refer to the same curve. For example, the strings `secp256r1`, `1.2.840.10045.3.1.7`, `NIST P-256`, and `X9.62 prime256v1` refer to the same curve. You can use the curve names to create parameter specifications for EC parameter generation with the `ECGenParameterSpec` class or the `NamedParameterSpec` class.

Recommended Curves

The following table lists the elliptic curves that are provided by the `SunEC` provider and are implemented using modern formulas and techniques.

Table 4-29 Recommended Curves Provided by the SunEC Provider

Curve Name	Object Identifier	Additional Names/Aliases
Ed25519	1.3.101.112	N/A
Ed448	1.3.101.113	N/A
secp256r1	1.2.840.10045.3.1.7	NIST P-256, X9.62 prime256v1
secp384r1	1.3.132.0.34	NIST P-384
secp521r1	1.3.132.0.35	NIST P-521

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-29 (Cont.) Recommended Curves Provided by the SunEC Provider

Curve Name	Object Identifier	Additional Names/Aliases
X25519	1.3.101.110	N/A
X448	1.3.101.111	N/A

Object Identifiers Associated with ECDSA Signatures

The following table lists object identifiers (OIDs) associated with ECDSA Signatures:

Table 4-30 OIDs associated with ECDSA Signatures

ECDSA Signature	OID
SHA1withECDSA	1.2.840.10045.4.1
SHA224withECDSA	1.2.840.10045.4.3.1
SHA256withECDSA	1.2.840.10045.4.3.2
SHA384withECDSA	1.2.840.10045.4.3.3
SHA512withECDSA	1.2.840.10045.4.3.4
SHA3-224withECDSA	2.16.840.1.101.3.4.3.9
SHA3-256withECDSA	2.16.840.1.101.3.4.3.10
SHA3-384withECDSA	2.16.840.1.101.3.4.3.11
SHA3-512withECDSA	2.16.840.1.101.3.4.3.12

The Apple Provider

The `Apple` provider implements a `java.security.KeyStore` that provides access to the macOS Keychain.

Algorithms

The following algorithms are available in the `Apple` provider:

Table 4-31 The Apple Provider Algorithm Name for Engine Classes

Engine	Algorithm Name(s)
`KeyStore`	**KeychainStore**: Contains private keys and certificates for the user's current keychain **KeychainStore-ROOT**: Contains certificates from the system root certificates keychain

The JdkLDAP Provider

The `JdkLDAP` provider replaces the LDAP CertStore implementation in the `SUN` provider.

Algorithms

The following algorithms are available in the `JdkLDAP` provider:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 4-32 The JdkLDAP Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`CertStore`	LDAP

The JdkSASL Provider

Algorithms

The following algorithms are available in the `JdkSASL` provider:

Table 4-33 The JdkSASL Provider Algorithm Names for Engine Classes

Engine	Algorithm Names
`SaslClient`	GSSAPI
`SaslServer`	GSSAPI

Copyright © 1993, 2026, Oracle and/or its affiliates.

5

PKCS#11 Reference Guide

The Java platform defines a set of programming interfaces for performing cryptographic operations. These interfaces are collectively known as the Java Cryptography Architecture (JCA) and the Java Cryptography Extension (JCE). See Java Cryptography Architecture (JCA) Reference Guide.

The cryptographic interfaces are provider-based. Specifically, applications talk to Application Programming Interfaces (APIs), and the actual cryptographic operations are performed in configured providers which adhere to a set of Service Provider Interfaces (SPIs). This architecture supports different provider implementations. Some providers may perform cryptographic operations in software; others may perform the operations on a hardware token (for example, on a smartcard device or on a hardware cryptographic accelerator).

The Cryptographic Token Interface Standard, PKCS#11, is produced by RSA Security and defines native programming interfaces to cryptographic tokens, such as hardware cryptographic accelerators and smartcards. Existing applications that use the JCA and JCE APIs can access native PKCS#11 tokens with the PKCS#11 provider. No modifications to the application are required. The only requirement is to properly configure the provider.

Although an application can make use of most PKCS#11 features using existing APIs, some applications might need more flexibility and capabilities. For example, an application might want to deal with smartcards being removed and inserted dynamically more easily. Or, a PKCS#11 token might require authentication for some non-key-related operations and therefore, the application must be able to log into the token without using keystore. The JCA gives applications greater flexibility in dealing with different providers.

This document describes how native PKCS#11 tokens can be configured into the Java platform for use by Java applications. It also describes how the JCA makes it easier for applications to deal with different types of providers, including PKCS#11 providers.

SunPKCS11 Provider

The SunPKCS11 provider, in contrast to most other providers, does not implement cryptographic algorithms itself. Instead, it acts as a bridge between the Java JCA and JCE APIs and the native PKCS#11 cryptographic API, translating the calls and conventions between the two.

This means that Java applications calling standard JCA and JCE APIs can, without modification, take advantage of algorithms offered by the underlying PKCS#11 implementations, such as, for example,

- Cryptographic smartcards,
- Hardware cryptographic accelerators, and
- High performance software implementations.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> Java SE only facilitates accessing native PKCS#11 implementations, it does not itself include a native PKCS#11 implementation. However, cryptographic devices such as Smartcards and hardware accelerators often come with software that includes a PKCS#11 implementation, which you need to install and configure according to manufacturer's instructions.

SunPKCS11 Requirements

The SunPKCS11 provider requires an implementation of PKCS#11 v2.20 or later to be installed on the system. This implementation must take the form of a shared-object library (`.so` on Linux) or dynamic-link library (`.dll` on Windows or `.dylib` on macOS). Consult your vendor documentation to find out if your cryptographic device includes such a PKCS#11 implementation, how to configure it, and what the name of the library file is.

The SunPKCS11 provider supports a number of algorithms, provided that the underlying PKCS#11 implementation offers them. The algorithms and their corresponding PKCS#11 mechanisms are listed in the table in SunPKCS11 Provider Supported Algorithms.

SunPKCS11 Configuration

The SunPKCS11 provider is in the module `jdk.crypto.cryptoki`. To use the provider, you must first install it statically or programmatically.

To install the provider statically, add the provider to the Java security properties file (*`java-home`*`/conf/security/java.security`).

> **Note**
>
> Properties in the `java.security` file are typically parsed only once. If you have modified any property in this file, restart your applications to ensure that the changes are properly reflected.

For example, here's a fragment of the `java.security` file that installs the SunPKCS11 provider with the configuration file `/opt/bar/cfg/pkcs11.cfg`.

```
# configuration for security providers 1-12 omitted
security.provider.13=SunPKCS11 /opt/bar/cfg/pkcs11.cfg
```

To install the provider dynamically, create an instance of the provider with the appropriate configuration file name and then install it. Here is an example.

```
String configName = "/opt/bar/cfg/pkcs11.cfg";
Provider p = Security.getProvider("SunPKCS11");
p = p.configure(configName);
Security.addProvider(p);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> Save the returned `Provider` object from the `configure` method, then add that object, as demonstrated in this example:
>
> ```
> p = p.configure(configName);
> Security.addProvider(p);
> ```
>
> Don't add the provider from which you called the `configure` method:
>
> ```
> p.configure(configName);
> Security.addProvider(p);
> ```
>
> If this provider cannot be configured in-place, then a new provider is created and returned. Therefore, always use the provider returned from the `configure` method.

To use more than one slot per PKCS#11 implementation, or to use more than one PKCS#11 implementation, simply repeat the installation for each with the appropriate configuration file. This will result in a SunPKCS11 provider instance for each slot of each PKCS#11 implementation.

The configuration file is a text file that contains entries in the following format:

`attribute=value`
The valid values for *attribute* and *value* are described in the table in this section:

The two mandatory attributes are `name` and `library`.
Here is a sample configuration file:

```
name = FooAccelerator
library = /opt/foo/lib/libpkcs11.so
```

Comments are denoted by lines starting with the # (number) symbol.

Table 5-1 Attributes in the PKCS#11 Provider Configuration File

Attribute	Value	Description
`allowLegacy`	Boolean value, default: `false`	For Cipher crypto service registrations, PKCS11 mechanisms supporting decryption but not encryption are considered legacy and will be disabled by default. Similarly, for Signature crypto service registrations, PKCS11 mechanisms supporting verification but not signing are considered legacy and will be disabled by default. If `true`, then the SunPKCS11 provider will not disable legacy mechanisms.
`allowSingleThreadedModules`	Boolean value, default: `true`	If `true`, then allow modules that only support single threaded access. Single threaded modules cannot be used safely from multiple PKCS#11 consumers in the same process, for example, when using Network Security Services (NSS) with SunPKCS11.
`attributes`	See Attributes Configuration	Specifies additional PKCS#11 attributes that should be set when creating PKCS#11 key objects. This makes it possible to accommodate tokens that require particular attributes.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-1 (Cont.) Attributes in the PKCS#11 Provider Configuration File

Attribute	Value	Description
`cipherTextStealingVariant`	One of `CS1`, `CS2`, or `CS3`	Specifies the CTS variant used by the PKCS#11 implementation, as described in Addendum to NIST Special Publication 800-38A, "Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode". For maximum interoperability, the SunPKCS11 provider uses the CS3 variant and will handle the data conversion if `CS1` or `CS2` is specified. If no variant is specified, the CKM_AES_CTS mechanism is disabled except for NSS whose implementation is known to be CS1.
`cleaner.longInterval`	Integer in milliseconds, default `60000`. The value must be greater than 1000 ms.	Specifies how often, in milliseconds. the cleaner thread should check for native references during non-busy periods, that is, the frequency that the cleaner thread checks the queue for native references. **Note** The cleaner thread will switch to the `cleaner.shortInterval` frequency if native PKCS11 references for cleaning are detected.
`cleaner.shortInterval`	Integer in milliseconds, default: `2000`. The value must be greater than 1000 ms.	Specifies how often, in milliseconds, native reference clearing should be performed during busy periods, that is, the frequency that the cleaner thread processes no-longer-needed native references in the queue to free up native memory. **Note** The cleaner thread will switch to the `cleaner.longInterval` frequency after 200 failed tries, that is, when no references are found in the queue.
`description`	Description of this provider instance	Specifies the string that the provider instance's `Provider.getInfo()` method returns. If no string is specified, then a default description is returned.
`destroyTokenAfterLogout`	Boolean value, default: `false`	If `true`, then when `java.security.AuthProvider.logout()` is called from the SunPKCS11 provider instance, the underlying token object will be destroyed and resources will be freed. This essentially renders the SunPKCS11 provider instance unusable after `logout()` calls. **Note** You should not add a SunPKCS11 provider with this attribute set to `true` to the system provider list because the provider object is not useable after `logout()` is called.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-1 (Cont.) Attributes in the PKCS#11 Provider Configuration File

Attribute	Value	Description
`disabledMechanisms`	Brace enclosed, whitespace-separated list of PKCS#11 mechanisms to disable	Specifies the list of PKCS#11 mechanisms that this provider instance should ignore. The provider ignores any mechanism listed, even if they are supported by the token and the SunPKCS11 provider. Specify the strings `SecureRandom` and `KeyStore` to disable those services. At most, you can specify one of `enabledMechanisms` or `disabledMechanisms`. If you specify neither, then the mechanisms enabled are those that are supported by both the SunPKCS11 provider (see SunPKCS11 Provider Supported Algorithms) and the PKCS#11 token.

`enabledMechanisms`	Brace enclosed, whitespace-separated list of PKCS#11 mechanisms to enable	Specifies the list PKCS#11 mechanisms that this provider instance should use, provided that they are supported by both the SunPKCS11 provider and PKCS#11 token. All other mechanisms are ignored. Each entry in the list is the name of a PKCS#11 mechanism. Here is an example that lists two PKCS#11 mechanisms.
		
		`enabledMechanisms = {`
		`  CKM_RSA_PKCS`
		`  CKM_RSA_PKCS_KEY_PAIR_GEN`
		`}`
		
		At most, you can specify one of `enabledMechanisms` or `disabledMechanisms`. If you specify neither, then the mechanisms enabled are those that are supported by both the SunPKCS11 provider (see SunPKCS11 Provider Supported Algorithms) and the PKCS#11 token.

`explicitCancel`	Boolean value, default: `true`	If `true`, then this attribute indicates that you must explicitly cancel operations.
`functionList`	Name of C function that returns the PKCS#11 function list, default: `C_GetFunctionList`	This option primarily exists for the deprecated `Secmod.Module.getProvider()` method.
`handleStartupErrors`	Possible values: `ignoreAll`, `ignoreMissingLibrary`, or `halt`; default: `halt`	Describes how to handle errors during startup.
`insertionCheckInterval`	Integer in milliseconds, default `2000`. The value must be greater than 100 ms.	Specifies how often to test for token insertion, in milliseconds, if no token is present.
`keyStoreCompatibilityMode`	Boolean value, default: `true`	If `true`, then this attribute indicates that P11Keystore is more tolerant of input parameters.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-1 (Cont.) Attributes in the PKCS#11 Provider Configuration File

Attribute	Value	Description
`library`	Pathname of PKCS#11 implementation	Specifies the the full pathname (including extension) of the PKCS#11 implementation; the format of the pathname is platform dependent. For example, `/opt/foo/lib/libpkcs11.so` might be the pathname of a PKCS#11 implementation on Linux while `C:\foo\mypkcs11.dll` might be the pathname on Windows or `/opt/local/lib/libpkcs11.dylib` on macOS. **Note**: • To configure Mozilla Network Security Services (NSS) as the PKCS#11 implementation, then set this attribute to the full pathname of the NSS softokn3 library. • Depending on your platform, you may have to set the environment variable `LD_LIBRARY_PATH` (Linux), `PATH` (Windows), or `DYLD_LIBRARY_PATH` (macOS) to include the enclosing directory to enable the operating system to locate the dependent libraries.
`name`	Name suffix of this provider instance	Specifies the string, which is concatenated with the prefix `SunPKCS11-` to produce this provider instance's name (that is, the string returned by its `Provider.getName()` method). For example, if the `name` attribute is `"FooAccelerator"`, then the provider instance's name will be `"SunPKCS11-FooAccelerator"`.
`nssArgs`	Quoted string	Specifies a special initialization argument string for the NSS soft token. This is used when using the NSS soft token directly without secmod mode.
`nssDbMode`	See Table 5-2	See Table 5-2
`nssLibraryDirectory`	See Table 5-2	See Table 5-2
`nssModule`	See Table 5-2	See Table 5-2
`nssNetscapeDbWorkaround`	See Table 5-2	See Table 5-2
`nssOptimizeSpace`	See Table 5-2	See Table 5-2
`nssSecmodDirectory`	See Table 5-2	See Table 5-2
`nssUseSecmod`	See Table 5-2	See Table 5-2
`omitInitialize`	Boolean value, default: `false`	If `true`, then omit the call to the `C_Initialize()` function. Use only if the PKCS#11 implementation has been initialized earlier with a `C_Initialize()` call.
`showInfo`	Boolean value, default: `false`	If `true`, then display provider information during start up. Provider information includes the provider's name and supported PKCS#11 mechanisms.
`slot`	Slot ID	Specifies the ID of the slot that this provider instance is to be associated with. For example, you would use `1` for the slot with the id `1` under PKCS#11. At most one of `slot` or `slotListIndex` may be specified. If neither is specified, the default is a `slotListIndex` of `0`.
`slotListIndex`	Slot index	Specifies the slot index that this provider instance is to be associated with. It is the index into the list of all slots returned by the PKCS#11 function `C_GetSlotList`. For example, `0` indicates the first slot in the list. At most one of `slot` or `slotListIndex` may be specified. If neither is specified, the default is a `slotListIndex` of `0`.
`useEcX963Encoding`	Boolean value, default: `false`	Indicates that the X9.63 encoding for EC points is used (`true`) or that the encoding is wrapped in an ASN.1 OctetString (`false`).

Copyright © 1993, 2026, Oracle and/or its affiliates.

Attributes Configuration

The attributes option allows you to specify additional PKCS#11 attributes that should be set when creating PKCS#11 key objects. By default, the SunPKCS11 provider only specifies mandatory PKCS#11 attributes when creating objects. For example, for RSA public keys it specifies the key type and algorithm (CKA_CLASS and CKA_KEY_TYPE) and the key values for RSA public keys (CKA_MODULUS and CKA_PUBLIC_EXPONENT). The PKCS#11 library you are using will assign implementation specific default values to the other attributes of an RSA public key, for example that the key can be used to encrypt and verify messages (CKA_ENCRYPT and CKA_VERIFY = true).

The `attributes` option can be used if you do not like the default values your PKCS#11 implementation assigns or if your PKCS#11 implementation does not support defaults and requires a value to be specified explicitly. Note that specifying attributes that your PKCS#11 implementation does not support or that are invalid for the type of key in question may cause the operation to fail at runtime.

The option can be specified zero or more times. The options are processed in the order specified in the configuration file. The `attributes` option has the format:

```
attributes(operation, keytype, keyalgorithm) = {
  name1 = value1
  [...]
}
```

Valid values for `operation` are:

- `generate`, for keys generated via a `KeyPairGenerator` or `KeyGenerator`
- `import`, for keys created via a `KeyFactory` or `SecretKeyFactory`. This also applies to Java software keys automatically converted to PKCS#11 key objects when they are passed to the initialization method of a cryptographic operation, for example `Signature.initSign()`.
- `*`, for keys created using either a generate or a create operation.

Valid values for `keytype` are `CKO_PUBLIC_KEY`, `CKO_PRIVATE_KEY`, and `CKO_SECRET_KEY`, for public, private, and secret keys, respectively, and `*` to match any type of key.

Valid values for `keyalgorithm` are one of the `CKK_xxx` constants from the PKCS#11 specification, or `*` to match keys of any algorithm. See SunPKCS11 Provider Supported Algorithms.

The attribute names and values are specified as a list of one or more name-value pairs. `name` must be a `CKA_xxx` constant from the PKCS#11 specification, for example `CKA_SENSITIVE`. `value` can be one of the following:

- A boolean value, `true` or `false`
- An integer, in decimal form (default) or in hexadecimal form if it begins with `0x`.
- `null`, indicating that this attribute should *not* be specified when creating objects.

If the `attributes` option is specified multiple times, the entries are processed in the order specified with the attributes aggregated and later attributes overriding earlier ones. For example, consider the following configuration file excerpt:

```
attributes(*,CKO_PRIVATE_KEY,*) = {
  CKA_SIGN = true
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
}

attributes(*,CKO_PRIVATE_KEY,CKK_DH) = {
  CKA_SIGN = null
}

attributes(*,CKO_PRIVATE_KEY,CKK_RSA) = {
  CKA_DECRYPT = true
}
```

The first entry says to specify `CKA_SIGN = true` for all private keys. The second option overrides that with `null` for Diffie-Hellman private keys, so the `CKA_SIGN` attribute will not specified for them at all. Finally, the third option says to also specify `CKA_DECRYPT = true` for RSA private keys. That means RSA private keys will have both `CKA_SIGN = true` and `CKA_DECRYPT = true` set.

There is also a special form of the `attributes` option. You can write `attributes = compatibility` in the configuration file. That is a shortcut for a whole set of attribute statements. They are designed to provider maximum compatibility with existing Java applications, which may expect, for example, all key components to be accessible and secret keys to be usable for both encryption and decryption. The `compatibility` attributes line can be used together with other `attributes` lines, in which case the same aggregation and overriding rules apply as described earlier.

Accessing Network Security Services (NSS)

Network Security Services (NSS) is a set of open source security libraries whose crypto APIs are based on PKCS#11 but it includes special features that are outside of the PKCS#11 standard. The SunPKCS11 provider includes code to interact with these NSS specific features, including several NSS specific configuration directives.

For best results, we recommend that you use the latest version of NSS available. It should be at least version 3.12.

The SunPKCS11 provider uses NSS specific code when any of the `nss` configuration directives described in Table 5-2 are used. In that case, the regular configuration commands `library`, `slot`, and `slotListIndex` cannot be used.

Table 5-2 NSS Attributes and Values

Attribute	Value	Description
`nssDbMode`	One of `readWrite`, `readOnly`, and `noDb`, default: `readWrite`	This directives determines how the NSS database is accessed. In read-write mode, full access is possible but only one process at a time should be accessing the databases. Read-only mode disallows modifications to the files. The noDb mode allows NSS to be used without database files purely as a cryptographic provider. It is not possible to create persistent keys using the PKCS11 KeyStore.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-2 (Cont.) NSS Attributes and Values

Attribute	Value	Description
`nssLibraryDirectory`	Directory containing the NSS and Netscape Portable Runtime (NSPR) libraries (which includes `libnss3.so`)	This is the full path name of the directory containing the NSS and NSPR libraries. It must be specified unless NSS has already been loaded and initialized by another component running in the same process as the Java VM. If this value is set, then `nssUseSecmod` is set to `true`. **Note**: Depending on your platform, you may have to set the environment variable `LD_LIBRARY_PATH` (Linux), `PATH` (Windows), or `DYLD_LIBRARY_PATH` (macOS) to include this directory to enable the operating system to locate the dependent libraries.
`nssModule`	One of `keystore`, `crypto`, `fips`, and `trustanchors`	NSS makes its functionality available using several different libraries and slots. This directive determines which of these modules is accessed by this instance of SunPKCS11. The `crypto` module is the default in `noDb` mode. It supports crypto operations without login but no persistent keys. The `fips` module is the default if the NSS `secmod.db` has been set to FIPS-140 compliant mode. In this mode, NSS restricts the available algorithms and the PKCS#11 attributes with which keys can be created. The `keystore` module is the default in other configurations. It supports persistent keys using the PKCS11 KeyStore, which are stored in the NSS DB files. This module requires login. The `trustanchors` module enables access to NSS trust anchor certificates via the PKCS11 KeyStore, if `secmod.db` has been configured to include the trust anchor library. If this value is set, then `nssUseSecmod` is set to `true`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-2 (Cont.) NSS Attributes and Values

Attribute	Value	Description
`nssNetscapeDbWorkaround`	Boolean value, default: `true`	If true, then the P11KeyStore specifies the `CKA_NETSCAPE_DB` attribute when creating private keys. This setting is only valid if `nssUseSecmod` is `true`
`nssOptimizeSpace`	Boolean value, default: `false`	Indicates that NSS favors performance (if `false`) or memory footprint (if `true`).
`nssSecmodDirectory`	The full path name of the directory containing the NSS configuration and key information (`secmod.db`, `key3.db`, and `cert8.db`)	This directive must be specified unless NSS has already been initialized by another component (see `nssLibraryDirectory`) or NSS is used without database files (see `nssDbMode`, noDb mode). If this value is set, then `nssUseSecmod` is set to `true`.
`nssUseSecmod`	Boolean value	If true, then NSS secmod mode is used. It's implicitly set to `true` if `nssLibraryDirectory`, `nssSecmodDirectory`, or `nssModule` is specified.

Example 5-1 SunPKCS11 Configuration Files for NSS

NSS as a pure cryptography provider

```
name = NSScrypto
nssLibraryDirectory = /opt/tests/nss/lib
nssDbMode = noDb
attributes = compatibility
```

NSS as a FIPS 140 compliant crypto token

```
name = NSSfips
nssLibraryDirectory = /opt/tests/nss/lib
nssSecmodDirectory = /opt/tests/nss/fipsdb
nssModule = fips
```

Troubleshooting PKCS#11

There could be issues with PKCS#11 which requires debugging. To show debug info about Library, Slots, Token, and Mechanism, add `showInfo=true` in the SunPKCS11 provider configuration file, which you specified statically or dynamically as described in SunPKCS11 Configuration.

For additional debugging info, users can start or restart the Java processes with one of the following options:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- For general SunPKCS11 provider debugging info:

  `-Djava.security.debug=sunpkcs11`

- For PKCS#11 keystore specific debugging info:

  `-Djava.security.debug=pkcs11keystore`

Disabling PKCS#11 Providers and/or Individual PKCS#11 Mechanisms

As part of the troubleshooting process, it could be helpful to temporarily disable a PKCS#11 provider or the specific mechanism of a given provider.

Please note that disabling a PKCS#11 provider, is only a temporary measure. By disabling the PKCS#11 provider, the provider is no longer available which can cause applications to break or have a performance impact. Once the issue has been identified, only that specific mechanism should remain disabled.

Disabling PKCS#11 Providers

Follow these steps to disable a PKCS#11 provider for all Java processes run with a particular Java installation:

1. Ensure that you have installed the PKCS#11 provider as described in SunPKCS11 Configuration. These steps assume the following:
   - The name of your PKCS#11 provider is `MyOwn`.
   - The name of your configuration file is *`java-home`*`/conf/security/myown.cfg` and contains the following:

     ```
     name = MyOwn
     description = A PKCS11 provider accessing a specific PKCS11 binary
     implementation
     library = Pathname of MyOwn PKCS11 implementation
     ```

   - You modified the Java security properties file (*`java-home`*`/conf/security/java.security`) as follows:

     ```
     #
     # List of providers and their preference orders:
     #
     security.provider.1=SUN
     security.provider.2=SunRsaSign
     security.provider.3=SunPKCS11 myOwn.cfg
     security.provider.4=SunEC
     security.provider.5=SunJSSE
     ...
     ```

2. To disable the `MyOwn` PKCS#11 provider statically for this Java installation, edit the Java security properties file and comment out the line that registers your provider, then renumber the preference order of the providers that follow it:

   ```
   #
   # List of providers and their preference orders:
   #
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
security.provider.1=SUN
security.provider.2=SunRsaSign
#security.provider.3=SunPKCS11 myOwn.cfg
security.provider.3=SunEC
security.provider.4=SunJSSE
...
```

3. To disable the `MyOwn` PKCS#11 provider dynamically for this Java installation, call the following in your application:

```
Security.removeProvider("SunPKCS11-MyOwn");
```

Disabling Specific Mechanisms

When an issue occurs in one of the mechanisms of PKCS#11, it can be resolved by disabling only that particular mechanism, rather than the entire PKCS#11 provider (do not forget to re-enable the PKCS#11 provider if it was disabled earlier).

> **Note**
>
> To disable the PKCS#11 SecureRandom implementation only, add `SecureRandom` to the list of disabled mechanisms in the SunPKCS11 provider configuration file, which you specified statically or dynamically as described in SunPKCS11 Configuration:
>
> ```
> disabledMechanisms = {
>     SecureRandom
> }
> ```

Application Developers

Java applications can use the existing JCA and JCE APIs to access PKCS#11 tokens through the SunPKCS11 provider.

Token Login

You can login to the keystore using a Personal Identification Number and perform PKCS#11 operations.

Certain PKCS#11 operations, such as accessing private keys, require a login using a Personal Identification Number, or PIN, before the operations can proceed. The most common type of operations that require login are those that deal with keys on the token. In a Java application, such operations often involve first loading the keystore. When accessing the PKCS#11 token as a keystore via the `java.security.KeyStore` class, you can supply the PIN in the password input parameter to the `load` method. The PIN will then be used by the SunPKCS11 provider for logging into the token. Here is an example.

```
char[] pin = ...;
KeyStore ks = KeyStore.getInstance("PKCS11");
ks.load(null, pin);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

This is fine for an application that treats PKCS#11 tokens as static keystores. For an application that wants to accommodate PKCS#11 tokens more dynamically, such as smartcards being inserted and removed, you can use the new `KeyStore.Builder` class. Here is an example of how to initialize the builder for a PKCS#11 keystore with a callback handler.

```
KeyStore.CallbackHandlerProtection chp =
    new KeyStore.CallbackHandlerProtection(new MyGuiCallbackHandler());
KeyStore.Builder builder =
    KeyStore.Builder.newInstance("PKCS11", null, chp);
```

For the SunPKCS11 provider, the callback handler must be able to satisfy a `PasswordCallback`, which is used to prompt the user for the PIN. Whenever the application needs access to the keystore, it uses the builder as follows.

```
KeyStore ks = builder.getKeyStore();
Key key = ks.getKey(alias, null);
```

The builder will prompt for a password as needed using the previously configured callback handler. The builder will prompt for a password only for the initial access. If the user of the application continues using the same Smartcard, the user will not be prompted again. If the user removes and inserts a different smartcard, the builder will prompt for a password for the new card.

Depending on the PKCS#11 token, there may be non-key-related operations that also require token login. Applications that use such operations can use the `java.security.AuthProvider` class. The `AuthProvider` class extends from `java.security.Provider` and defines methods to perform login and logout operations on a provider, as well as to set a callback handler for the provider to use.

For the SunPKCS11 provider, the callback handler must be able to satisfy a `PasswordCallback`, which is used to prompt the user for the PIN.

Here is an example of how an application might use an `AuthProvider` to log into the token. (Note that you must configure the SunPKCS11 provider before using it.)

```
    Provider p = Security.getProvider("SunPKCS11");
    AuthProvider aprov = (AuthProvider)p.configure(<provider configuration
file>);
    aprov.login(subject, new MyGuiCallbackHandler());
```

Token Keys

Java `Key` objects may or may not contain actual key material.

- A software `Key` object does contain the actual key material and allows access to that material.
- An unextractable key on a secure token (such as a smartcard) is represented by a Java `Key` object that does not contain the actual key material. The `Key` object only contains a reference to the actual key.

Applications and providers must use the correct interfaces to represent these different types of `Key` objects. Software `Key` objects (or any `Key` object that has access to the actual key material) should implement the interfaces in the `java.security.interfaces` and `javax.crypto.interfaces` packages (such as `DSAPrivateKey`). Key objects representing unextractable token keys should only implement the relevant generic interfaces in the

Copyright © 1993, 2026, Oracle and/or its affiliates.

`java.security` and `javax.crypto` packages (`PrivateKey`, `PublicKey`, or `SecretKey`). Identification of the algorithm of a key should be performed using the `Key.getAlgorithm()` method.

Note that a `Key` object for an unextractable token key can only be used by the provider associated with that token.

Delayed Provider Selection

Java cryptography `getInstance()` methods, such as `Cipher.getInstance("AES")`, return the implementation from the first provider that implemented the requested algorithm. However, the JDK delays the selection of the provider until the relevant initialization method is called. The initialization method accepts a `Key` object and can determine at that point which provider can accept the specified `Key` object. This ensures that the selected provider can use the specified `Key` object. (If an application attempts to use a `Key` object for an unextractable token key with a provider that only accepts software key objects, then the provider throws an `InvalidKeyException`. This is an issue for the `Cipher`, `KeyAgreement`, `Mac`, and `Signature` classes.) The following represents the affected initialization methods.

- `Cipher.init(..., Key key, ...)`
- `KeyAgreement.init(Key key, ...)`
- `Mac.init(Key key, ...)`
- `Signature.initSign(PrivateKey privateKey)`

> **Note**
>
> Once the provider is selected, for example, after the first initialization call, the JDK won't switch to a different provider for subsequent initialization calls. To reselect a provider based on a specific `Key` object, call `getInstance()` to get a new instance, and then call this instance's initialization method with the `Key` object instead of reusing the older, already-initialized instance.

Although this delayed provider selection is hidden from the application, it does affect the behavior of the `getProvider()` method for `Cipher`, `KeyAgreement`, `Mac`, and `Signature`. If `getProvider()` is called *before* the initialization operation has occurred (and therefore before provider selection has occurred), then the first provider that supports the requested algorithm is returned. This may not be the same provider as the one selected *after* the initialization method is called. If `getProvider()` is called *after* the initialization operation has occurred, then the actual selected provider is returned. It is recommended that applications only call `getProvider()` after they have called the relevant initialization method.

In addition to `getProvider()`, the following additional methods are similarly affected.

- `Cipher.getBlockSize`
- `Cipher.getExemptionMechanism`
- `Cipher.getIV`
- `Cipher.getOutputSize`
- `Cipher.getParameters`
- `Mac.getMacLength`

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `Signature.getParameters`
- `Signature.setParameter`

JAAS KeyStoreLoginModule

The JDK comes with a JAAS keystore login module, `KeyStoreLoginModule`, that allows an application to authenticate using its identity in a specified keystore. After authentication, the application would acquire its principal and credentials information (certificate and private key) from the keystore. By using this login module and configuring it to use a PKCS#11 token as a keystore, the application can acquire this information from a PKCS#11 token.

Use the following options to configure the `KeyStoreLoginModule` to use a PKCS#11 token as the keystore.

- `keyStoreURL="NONE"`
- `keyStoreType="PKCS11"`
- `keyStorePasswordURL=`*`some_pin_url`*

where

some_pin_url
The location of the PIN. If the `keyStorePasswordURL` option is omitted, then the login module will get the PIN via the application's callback handler, supplying it with a `PasswordCallback` . Here is an example of a configuration file that uses a PKCS#11 token as a keystore.

```
other {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL="NONE"
  keyStoreType="PKCS11"
  keyStorePasswordURL="file:/home/joe/scpin";
};
```

If more than one SunPKCS11 provider has been configured dynamically or in the `java.security` security properties file, you can use the `keyStoreProvider` option to target a specific provider instance. The argument to this option is the name of the provider. For the SunPKCS11 provider, the provider name is of the form `SunPKCS11-`*`TokenName`*, where *`TokenName`* is the name suffix that the provider instance has been configured with, as detailed in Table 5-1. For example, the following configuration file names the PKCS#11 provider instance with name suffix `SmartCard`.

```
other {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL="NONE"
  keyStoreType="PKCS11"
  keyStorePasswordURL="file:/home/joe/scpin"
  keyStoreProvider="SunPKCS11-SmartCard";
};
```

Some PKCS#11 tokens support login via a protected authentication path. For example, a smartcard may have a dedicated PIN-pad to enter the pin. Biometric devices will also have their own means to obtain authentication information. If the PKCS#11 token has a protected

Copyright © 1993, 2026, Oracle and/or its affiliates.

authentication path, then use the `protected=true` option and omit the `keyStorePasswordURL` option. Here is an example of a configuration file for such a token.

```
other {
    com.sun.security.auth.module.KeyStoreLoginModule required
        keyStoreURL="NONE"
  keyStoreType="PKCS11"
  protected=true;
};
```

Tokens as JSSE Keystore and Trust Stores

To use PKCS#11 tokens as JSSE keystores or trust stores, the JSSE application can use the APIs described in Token Login to instantiate a `KeyStore` that is backed by a PKCS#11 token and pass it to its key manager and trust manager. The JSSE application will then have access to the keys on the token.

JSSE also supports configuring the use of keystores and trust stores via system properties, as described in the Java Secure Socket Extension (JSSE) Reference Guide. To use a PKCS#11 token as a keystore or trust store, set the `javax.net.ssl.keyStoreType` and `javax.net.ssl.trustStoreType` system properties, respectively, to "PKCS11", and set the `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` system properties, respectively, to `NONE`. To specify the use of a specific provider instance, use the `javax.net.ssl.keyStoreProvider` and `javax.net.ssl.trustStoreProvider` system properties (for example, "SunPKCS11-SmartCard").

Using keytool and jarsigner with PKCS#11 Tokens

If the SunPKCS11 provider has been configured in the `java.security` security properties file (located in the `$JAVA_HOME/conf/security` directory of the Java runtime), then keytool and jarsigner can be used to operate on the PKCS#11 token by specifying the following options.

- `-keystore NONE`
- `-storetype PKCS11`

Here an example of a command to list the contents of the configured PKCS#11 token.

```
keytool -keystore NONE -storetype PKCS11 -list
```

The PIN can be specified using the `-storepass` option. If none has been specified, then `keytool` and `jarsigner` will prompt for the token PIN. If the token has a protected authentication path (such as a dedicated PIN-pad or a biometric reader), then the `-protected` option must be specified, and no password options can be specified.

If more than one SunPKCS11 provider has been configured in the `java.security` security properties file, you can use the `-providerName` option to target a specific provider instance. The argument to this option is the name of the provider.

- `-providerName` *`providerName`*

For the SunPKCS11 provider, *`providerName`* is of the form `SunPKCS11-`*`TokenName`* where:

Copyright © 1993, 2026, Oracle and/or its affiliates.

TokenName
The name suffix that the provider instance has been configured with, as detailed in Table 5-1. For example, the following command lists the contents of the PKCS#11 keystore provider instance with name suffix `SmartCard`.

```
keytool -keystore NONE -storetype PKCS11 \
    -providerName SunPKCS11-SmartCard \
    -list
```

If the SunPKCS11 provider has not been configured in the `java.security` security properties file, you can use the following options to instruct `keytool` and `jarsigner` to install the provider dynamically.

- `-providerClass sun.security.pkcs11.SunPKCS11`
- `-providerArg` *`ConfigFilePath`*

ConfigFilePath
The path to the token configuration file. Here is an example of a command to list a PKCS#11 keystore when the SunPKCS11 provider has not been configured in the `java.security` file.

```
keytool -keystore NONE -storetype PKCS11 \
     -providerClass sun.security.pkcs11.SunPKCS11 \
     -providerArg /foo/bar/token.config \
     -list
```

> **Note**
>
> Sometimes the hardware token is too small to store the certificates. You can use the `jarsigner` tool's `-certchain` option to load them from an external file.

Signing JAR File with jdk.security.jarsigner API and PKCS#11 Token

To sign a JAR file with the `jdk.security.jarsigner` API and a PKCS#11 token as a keystore, follow these steps:

1. Access the PKCS#11 token's keystore as described in Token Login:

   ```
   char[] pin = ...;
   KeyStore ks = KeyStore.getInstance("PKCS11");
   ks.load(null, pin);
   KeyStore ks = builder.getKeyStore();
   Key key = ks.getKey(alias, null);
   ```

2. Create a `JarSigner` object with the `JarSigner.Builder(Keystore.PrivateKeyEntry)` constructor:

   ```
   JarSigner mySigner = JarSigner.Builder(key).build();
   ```

3. Sign the JAR file:

   ```
   try (ZipFile in = new ZipFile(inputFile);
   FileOutputStream out = new FileOutputStream(outputFile)) {
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            mySigner.sign(in, out);
        }
```

Provider Developers

The `java.security.Provider` class enables provider developers to more easily support PKCS#11 tokens and cryptographic services through provider services and parameter support.

See Example Provider for an example of a simple provider designed to demonstrate provider services and parameter support.

Provider Services

For each service implemented by the provider, there must be a property whose name is the type of service (`Cipher`, `Signature`, etc), followed by a period and the name of the algorithm to which the service applies. The property value must specify the fully qualified name of the class implementing the service. Here is an example of a provider setting `KeyAgreement.DiffieHellman` property to have the value `com.sun.crypto.provider.DHKeyAgreement`.

```
put("KeyAgreement.DiffieHellman", "com.sun.crypto.provider.DHKeyAgreement")
```

The public static nested class `Provider.Service` encapsulates the properties of a provider service (including its type, attributes, algorithm name, and algorithm aliases). Providers can instantiate `Provider.Service` objects and register them by calling the `Provider.putService()` method. This is equivalent to creating a `Property` entry and calling the `Provider.put()` method. Note that legacy `Property` entries registered via `Provider.put` are still supported.

Here is an example of a provider creating a `Service` object with the `KeyAgreement` type, for the `DiffieHellman` algorithm, implemented by the class `com.sun.crypto.provider.DHKeyAgreement`.

```
Service s = new Service(this, "KeyAgreement", "DiffieHellman",
    "com.sun.crypto.provider.DHKeyAgreement", null, null);
putService(s);
```

Using `Provider.Service` objects instead of legacy `Property` entries has a couple of major benefits. One benefit is that it allows the provider to have greater flexibility when Instantiating Engine Classes. Another benefit is that it allows the provider to test Parameter Support. These features are discussed in detail next.

Instantiating Engine Classes

By default, the Java Cryptography framework looks up the provider property for a particular service and directly instantiates the engine class registered for that property. A provider can to override this behavior and instantiate the engine class for the requested service itself.

To override the default behavior, the provider overrides the `Provider.Service.newInstance()` method to add its custom behavior. For example, the provider might call a custom constructor, or might perform initialization using information not accessible outside the provider (or that are only known by the provider).

Copyright © 1993, 2026, Oracle and/or its affiliates.

Parameter Support

The Java Cryptography framework may attempt a fast check to determine whether a provider's service implementation can use an application-specified parameter. To perform this fast check, the framework calls `Provider.Service.supportsParameter()`.

The framework relies on this fast test during delayed provider selection (see Delayed Provider Selection). When an application invokes an initialization method and passes it a `Key` object, the framework asks an underlying provider whether it supports the object by calling its `Service.supportsParameter()` method. If `supportsParameter()` returns `false`, the framework can immediately remove that provider from consideration. If `supportsParameter()` returns `true`, the framework passes the `Key` object to that provider's initialization engine class implementation. A provider that requires software `Key` objects should override this method to return `false` when it is passed non-software keys. Likewise, a provider for a PKCS#11 token that contains unextractable keys should only return `true` for `Key` objects that it created, and which therefore correspond to the keys on its respective token.

> **Note**
>
> The default implementation of `supportsParameter()` returns `true`. This allows existing providers to work without modification. However, because of this lenient default implementation, the framework must be prepared to catch exceptions thrown by providers that reject the `Key` object inside their initialization engine class implementations. The framework treats these cases the same as when `supportsParameter()` returns `false`.

Parameter Support

The Java Cryptography framework may attempt a fast check to determine whether a provider's service implementation can use an application-specified parameter. To perform this fast check, the framework calls `Provider.Service.supportsParameter()`.

The framework relies on this fast test during delayed provider selection (see Delayed Provider Selection). When an application invokes an initialization method and passes it a `Key` object, the framework asks an underlying provider whether it supports the object by calling its `Service.supportsParameter()` method. If `supportsParameter()` returns `false`, the framework can immediately remove that provider from consideration. If `supportsParameter()` returns `true`, the framework passes the `Key` object to that provider's initialization engine class implementation. A provider that requires software `Key` objects should override this method to return `false` when it is passed non-software keys. Likewise, a provider for a PKCS#11 token that contains unextractable keys should only return `true` for `Key` objects that it created, and which therefore correspond to the keys on its respective token.

> **Note**
>
> The default implementation of `supportsParameter()` returns `true`. This allows existing providers to work without modification. However, because of this lenient default implementation, the framework must be prepared to catch exceptions thrown by providers that reject the `Key` object inside their initialization engine class implementations. The framework treats these cases the same as when `supportsParameter()` returns `false`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

SunPKCS11 Provider Supported Algorithms

Table 5-3 lists the Java algorithms supported by the SunPKCS11 provider and corresponding PKCS#11 mechanisms needed to support them. When multiple mechanisms are listed, they are given in the order of preference and any one of them is sufficient.

> **Note**
>
> SunPKCS11 can be instructed to ignore mechanisms by using the `disabledMechanisms` and `enabledMechanisms` configuration directives (see SunPKCS11 Configuration).

For Elliptic Curve mechanisms, the SunPKCS11 provider will only use keys that use the `namedCurve` choice as encoding for the parameters and only allow the uncompressed point format. The SunPKCS11 provider assumes that a token supports all standard named domain parameters.

> **Note**
>
> For Elliptic Curve (EC) names, the SunPKCS11 provider supports any EC name that the SunEC provider supports as long as the token supports it; see Supported Elliptic Curve Names in The SunEC Provider.

HKDF algorithms in SunPKCS11 rely on the CKM_HKDF_DATA and CKM_HKDF_DERIVE native mechanisms in addition to the required mechanisms from the PKCS#11 library. If only one of these two native mechanisms is supported, then a runtime failure will occur upon invoking the corresponding method: `KDF.deriveData(AlgorithmParameterSpec)` for CKM_HKDF_DATA and `KDF.deriveKey(String, AlgorithmParameterSpec)` for CKM_HKDF_DERIVE.

> **Note**
>
> If the PKCS#11 library does not support the CKM_CONCATENATE_BASE_AND_DATA, CKM_CONCATENATE_BASE_AND_KEY, and CKM_CONCATENATE_DATA_AND_BASE native mechanisms, then a runtime error will occur when the concatenation of the forms *key-data*, *key-key* or *data-key* is attempted. See the `HKDFParameterSpec.Builder.addIKM` and `HKDFParameterSpec.Builder.addSalt` methods in the `HKDFParameterSpec.Builder` class and the `HKDFParameterSpec` interface in the JavaDoc API documentation for more information about key concatenation.

See PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0 for more details about the PKCS#11 mechanisms in the following table.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-3 Java Algorithms Supported by the SunPKCS11 Provider

Java Algorithm	PKCS#11 Mechanisms
Cipher.AES_128/CBC/NoPadding	CKM_AES_CBC
Cipher.AES_128/CTS/NoPadding	CKM_AES_CTS
Cipher.AES_128/ECB/NoPadding	CKM_AES_ECB
Cipher.AES_128/GCM/NoPadding	CKM_AES_GCM
Cipher.AES_192/CBC/NoPadding	CKM_AES_CBC
Cipher.AES_192/CTS/NoPadding	CKM_AES_CTS
Cipher.AES_192/ECB/NoPadding	CKM_AES_ECB
Cipher.AES_192/GCM/NoPadding	CKM_AES_GCM
Cipher.AES_256/CBC/NoPadding	CKM_AES_CBC
Cipher.AES_256/CTS/NoPadding	CKM_AES_CTS
Cipher.AES_256/ECB/NoPadding	CKM_AES_ECB
Cipher.AES_256/GCM/NoPadding	CKM_AES_GCM
Cipher.AES/CBC/NoPadding	CKM_AES_CBC
Cipher.AES/CBC/PKCS5Padding	CKM_AES_CBC_PAD, CKM_AES_CBC
Cipher.AES/CTR/NoPadding	CKM_AES_CTR
Cipher.AES/CTS/NoPadding	CKM_AES_CTS
Cipher.AES/ECB/NoPadding	CKM_AES_ECB
Cipher.AES/ECB/PKCS5Padding	CKM_AES_ECB
Cipher.AES/GCM/NoPadding	CKM_AES_GCM
Cipher.ARCFOUR	CKM_RC4
Cipher.Blowfish/CBC/NoPadding	CKM_BLOWFISH_CBC
Cipher.Blowfish/CBC/PKCS5Padding	CKM_BLOWFISH_CBC
Cipher.ChaCha20-Poly1305	CKM_CHACHA20_POLY1305
Cipher.DES/CBC/NoPadding	CKM_DES_CBC
Cipher.DES/CBC/PKCS5Padding	CKM_DES_CBC_PAD, CKM_DES_CBC
Cipher.DES/ECB/NoPadding	CKM_DES_ECB
Cipher.DES/ECB/PKCS5Padding	CKM_DES_ECB
Cipher.DESede/CBC/NoPadding	CKM_DES3_CBC
Cipher.DESede/CBC/PKCS5Padding	CKM_DES3_CBC_PAD, CKM_DES3_CBC
Cipher.DESede/ECB/NoPadding	CKM_DES3_ECB
Cipher.DESede/ECB/PKCS5Padding	CKM_DES3_ECB
Cipher.PBEWithHmacSHA1AndAES_128	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA_1_HMAC required)
Cipher.PBEWithHmacSHA224AndAES_128	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA224_HMAC required)
Cipher.PBEWithHmacSHA256AndAES_128	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA256_HMAC required)
Cipher.PBEWithHmacSHA384AndAES_128	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA384_HMAC required)

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-3 (Cont.) Java Algorithms Supported by the SunPKCS11 Provider

Java Algorithm	PKCS#11 Mechanisms
Cipher.PBEWithHmacSHA512AndAES_128	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA512_HMAC required)
Cipher.PBEWithHmacSHA1AndAES_256	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA_1_HMAC required)
Cipher.PBEWithHmacSHA224AndAES_256	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA224_HMAC required)
Cipher.PBEWithHmacSHA256AndAES_256	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA256_HMAC required)
Cipher.PBEWithHmacSHA384AndAES_256	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA384_HMAC required)
Cipher.PBEWithHmacSHA512AndAES_256	CKM_AES_CBC_PAD, CKM_AES_CBC (CKM_PKCS5_PBKD2 and CKM_SHA512_HMAC required)
Cipher.RSA/ECB/NoPadding	CKM_RSA_X_509
Cipher.RSA/ECB/PKCS1Padding	CKM_RSA_PKCS
KeyAgreement.DiffieHellman	CKM_DH_PKCS_DERIVE
KeyAgreement.ECDH	CKM_ECDH1_DERIVE
KeyFactory.DiffieHellman	Any supported Diffie-Hellman mechanism
KeyFactory.DSA	Any supported DSA mechanism
KeyFactory.EC	Any supported EC mechanism
KeyFactory.RSA	Any supported RSA mechanism
KeyGenerator.AES	CKM_AES_KEY_GEN
KeyGenerator.ARCFOUR	CKM_RC4_KEY_GEN
KeyGenerator.Blowfish	CKM_BLOWFISH_KEY_GEN
KeyGenerator.ChaCha20	CKM_CHACHA20_KEY_GEN
KeyGenerator.DES	CKM_DES_KEY_GEN
KeyGenerator.DESede	CKM_DES3_KEY_GEN
KeyGenerator.HmacMD5	CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA1	CKM_SHA_1_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA224	CKM_SHA224_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA256	CKM_SHA256_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA3-224	CKM_SHA3_224_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA3-256	CKM_SHA3_256_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA3-384	CKM_SHA3_384_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA3-512	CKM_SHA3_512_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-3 (Cont.) Java Algorithms Supported by the SunPKCS11 Provider

Java Algorithm	PKCS#11 Mechanisms
KeyGenerator.HmacSHA384	CKM_SHA384_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA512	CKM_SHA512_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA512/224	CKM_SHA512_224_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyGenerator.HmacSHA512/256	CKM_SHA512_256_KEY_GEN, CKM_GENERIC_SECRET_KEY_GEN
KeyPairGenerator.DiffieHellman	CKM_DH_PKCS_KEY_PAIR_GEN
KeyPairGenerator.DSA	CKM_DSA_KEY_PAIR_GEN
KeyPairGenerator.EC	CKM_EC_KEY_PAIR_GEN
KeyPairGenerator.RSA	CKM_RSA_PKCS_KEY_PAIR_GEN
KeyStore.PKCS11	Always available
KDF.HKDF-SHA256	CKM_HKDF_DATA, CKM_HKDF_DERIVE (CKM_SHA256_HMAC required)
KDF.HKDF-SHA384	CKM_HKDF_DATA, CKM_HKDF_DERIVE (CKM_SHA384_HMAC required)
KDF.HKDF-SHA512	CKM_HKDF_DATA, CKM_HKDF_DERIVE (CKM_SHA512_HMAC required)
Mac.HmacMD5	CKM_MD5_HMAC
Mac.HmacPBESHA1	CKM_PBA_SHA1_WITH_SHA1_HMAC (CKM_SHA_1_HMAC required)
Mac.HmacPBESHA224	CKM_NSS_PKCS12_PBE_SHA224_HMAC_KEY_GEN (CKM_SHA224_HMAC required)
Mac.HmacPBESHA256	CKM_NSS_PKCS12_PBE_SHA256_HMAC_KEY_GEN (CKM_SHA256_HMAC required)
Mac.HmacPBESHA384	CKM_NSS_PKCS12_PBE_SHA384_HMAC_KEY_GEN (CKM_SHA384_HMAC required)
Mac.HmacPBESHA512	CKM_NSS_PKCS12_PBE_SHA512_HMAC_KEY_GEN (CKM_SHA512_HMAC required))
Mac.HmacSHA1	CKM_SHA_1_HMAC
Mac.HmacSHA224	CKM_SHA224_HMAC
Mac.HmacSHA256	CKM_SHA256_HMAC
Mac.HmacSHA3-224	CKM_SHA3_224_HMAC
Mac.HmacSHA3-256	CKM_SHA3_256_HMAC
Mac.HmacSHA3-384	CKM_SHA3_384_HMAC
Mac.HmacSHA3-512	CKM_SHA3_512_HMAC
Mac.HmacSHA384	CKM_SHA384_HMAC
Mac.HmacSHA512	CKM_SHA512_HMAC
Mac.HmacSHA512/224	CKM_SHA512_224_HMAC
Mac.HmacSHA512/256	CKM_SHA512_256_HMAC
MessageDigest.MD2	CKM_MD2
MessageDigest.MD5	CKM_MD5
MessageDigest.SHA-224	CKM_SHA224
MessageDigest.SHA-256	CKM_SHA256
MessageDigest.SHA-384	CKM_SHA384

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-3 (Cont.) Java Algorithms Supported by the SunPKCS11 Provider

Java Algorithm	PKCS#11 Mechanisms
MessageDigest.SHA-512	CKM_SHA512
MessageDigest.SHA-512/224	CKM_SHA512_224
MessageDigest.SHA-512/256	CKM_SHA512_256
MessageDigest.SHA1	CKM_SHA_1
MessageDigest.SHA3-224	CKM_SHA3_224
MessageDigest.SHA3-256	CKM_SHA3_256
MessageDigest.SHA3-384	CKM_SHA3_384
MessageDigest.SHA3-512	CKM_SHA3_512
SecretKeyFactory.AES	CKM_AES_CBC
SecretKeyFactory.ARCFOUR	CKM_RC4
SecretKeyFactory.Blowfish	CKM_BLOWFISH_CBC
SecretKeyFactory.ChaCha20	CKM_CHACHA20_POLY1305
SecretKeyFactory.DES	CKM_DES_CBC
SecretKeyFactory.DESede	CKM_DES3_CBC
SecretKeyFactory.Generic	CKM_GENERIC_SECRET_KEY_GEN
SecretKeyFactory.PBKDF2WithHmacSHA1	CKM_PKCS5_PBKD2 (CKM_SHA_1_HMAC required)
SecretKeyFactory.PBKDF2WithHmacSHA224	CKM_PKCS5_PBKD2 (CKM_SHA224_HMAC required)
SecretKeyFactory.PBKDF2WithHmacSHA256	CKM_PKCS5_PBKD2 (CKM_SHA256_HMAC required)
SecretKeyFactory.PBKDF2WithHmacSHA384	CKM_PKCS5_PBKD2 (CKM_SHA384_HMAC required)
SecretKeyFactory.PBKDF2WithHmacSHA512	CKM_PKCS5_PBKD2 (CKM_SHA512_HMAC required)
SecureRandom.PKCS11	CK_TOKEN_INFO has the CKF_RNG bit set
Signature.MD2withRSA	CKM_MD2_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.MD5withRSA	CKM_MD5_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.NONEwithDSA	CKM_DSA
Signature.NONEwithECDSA	CKM_ECDSA
Signature.RSASSA-PSS	CKM_RSA_PKCS_PSS
Signature.SHA1withDSA	CKM_DSA_SHA1, CKM_DSA
Signature.SHA1withECDSA	CKM_ECDSA_SHA1, CKM_ECDSA
Signature.SHA1withRSA	CKM_SHA1_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA1withRSASSA-PSS	CKM_SHA1_RSA_PKCS_PSS
Signature.SHA224withDSA	CKM_DSA_SHA224
Signature.SHA224withDSAinP1363Format	CKM_DSA_SHA224
Signature.SHA224withECDSA	CKM_ECDSA_SHA224, CKM_ECDSA
Signature.SHA224withECDSAinP1363Format	CKM_ECDSA_SHA224, CKM_ECDSA
Signature.SHA224withRSA	CKM_SHA224_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA224withRSASSA-PSS	CKM_SHA224_RSA_PKCS_PSS

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-3 (Cont.) Java Algorithms Supported by the SunPKCS11 Provider

Java Algorithm	PKCS#11 Mechanisms
Signature.SHA256withDSA	CKM_DSA_SHA256
Signature.SHA256withDSAinP1363Format	CKM_DSA_SHA256
Signature.SHA256withECDSA	CKM_ECDSA_SHA256, CKM_ECDSA
Signature.SHA256withECDSAinP1363Format	CKM_ECDSA_SHA256, CKM_ECDSA
Signature.SHA256withRSA	CKM_SHA256_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA256withRSASSA-PSS	CKM_SHA256_RSA_PKCS_PSS
Signature.SHA3-224withDSA	CKM_DSA_SHA3_224
Signature.SHA3-224withDSAinP1363Format	CKM_DSA_SHA3_224
Signature.SHA3-224withECDSA	CKM_ECDSA_SHA3_224, CKM_ECDSA
Signature.SHA3-224withECDSAinP1363Format	CKM_ECDSA_SHA3_224, CKM_ECDSA
Signature.SHA3-224withRSA	CKM_SHA3_224_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA3-224withRSASSA-PSS	CKM_SHA3_224_RSA_PKCS_PSS
Signature.SHA3-256withDSA	CKM_DSA_SHA3_256
Signature.SHA3-256withDSAinP1363Format	CKM_DSA_SHA3_256
Signature.SHA3-256withECDSA	CKM_ECDSA_SHA3_256, CKM_ECDSA
Signature.SHA3-256withECDSAinP1363Format	CKM_ECDSA_SHA3_256, CKM_ECDSA
Signature.SHA3-256withRSA	CKM_SHA3_256_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA3-256withRSASSA-PSS	CKM_SHA3_256_RSA_PKCS_PSS
Signature.SHA3-384withDSA	CKM_DSA_SHA3_384
Signature.SHA3-384withDSAinP1363Format	CKM_DSA_SHA3_384
Signature.SHA3-384withECDSA	CKM_ECDSA_SHA3_384, CKM_ECDSA
Signature.SHA3-384withECDSAinP1363Format	CKM_ECDSA_SHA3_384, CKM_ECDSA
Signature.SHA3-384withRSA	CKM_SHA3_384_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA3-384withRSASSA-PSS	CKM_SHA3_384_RSA_PKCS_PSS
Signature.SHA3-512withDSA	CKM_DSA_SHA3_512
Signature.SHA3-512withDSAinP1363Format	CKM_DSA_SHA3_512
Signature.SHA3-512withECDSA	CKM_ECDSA_SHA3_512, CKM_ECDSA
Signature.SHA3-512withECDSAinP1363Format	CKM_ECDSA_SHA3_512, CKM_ECDSA
Signature.SHA3-512withRSA	CKM_SHA3_512_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA3-512withRSASSA-PSS	CKM_SHA3_512_RSA_PKCS_PSS
Signature.SHA384withDSA	CKM_DSA_SHA384
Signature.SHA384withDSAinP1363Format	CKM_DSA_SHA384
Signature.SHA384withECDSA	CKM_ECDSA_SHA384, CKM_ECDSA
Signature.SHA384withECDSAinP1363Format	CKM_ECDSA_SHA384, CKM_ECDSA
Signature.SHA384withRSA	CKM_SHA384_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA384withRSASSA-PSS	CKM_SHA384_RSA_PKCS_PSS
Signature.SHA512withDSA	CKM_DSA_SHA512
Signature.SHA512withDSAinP1363Format	CKM_DSA_SHA512

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 5-3 (Cont.) Java Algorithms Supported by the SunPKCS11 Provider

Java Algorithm	PKCS#11 Mechanisms
Signature.SHA512withECDSA	CKM_ECDSA_SHA512, CKM_ECDSA
Signature.SHA512withECDSAinP1363Format	CKM_ECDSA_SHA512, CKM_ECDSA
Signature.SHA512withRSA	CKM_SHA512_RSA_PKCS, CKM_RSA_PKCS, CKM_RSA_X_509
Signature.SHA512withRSASSA-PSS	CKM_SHA512_RSA_PKCS_PSS

SunPKCS11 Provider KeyStore Requirements

The following describes the requirements placed by the SunPKCS11 provider's KeyStore implementation on the underlying native PKCS#11 library.

> **Note**
>
> Changes may be made in future releases to maximize interoperability with as many existing PKCS#11 libraries as possible.

Read-Only Access

To map existing objects stored on a PKCS#11 token to KeyStore entries, the SunPKCS11 provider's KeyStore implementation performs the following operations.

1. A search for all private key objects on the token is performed by calling `C_FindObjects[Init|Final]`. The search template includes the following attributes:
   - CKA_TOKEN = true
   - CKA_CLASS = CKO_PRIVATE_KEY
2. A search for all certificate objects on the token is performed by calling `C_FindObjects[Init|Final]`. The search template includes the following attributes:
   - CKA_TOKEN = true
   - CKA_CLASS = CKO_CERTIFICATE
3. Each private key object is matched with its corresponding certificate by retrieving their respective CKA_ID attributes. A matching pair must share the same unique CKA_ID. For each matching pair, the certificate chain is built by following the issuer->subject path. From the end entity certificate, a call for `C_FindObjects[Init|Final]` is made with a search template that includes the following attributes:
   - CKA_TOKEN = true
   - CKA_CLASS = CKO_CERTIFICATE
   - CKA_SUBJECT = [DN of certificate issuer]

   This search is continued until either no certificate for the issuer is found, or until a self-signed certificate is found. If more than one certificate is found the first one is used.

   Once a private key and certificate have been matched (and its certificate chain built), the information is stored in a private key entry with the CKA_LABEL value from end entity certificate as the KeyStore alias.

Copyright © 1993, 2026, Oracle and/or its affiliates.

If the end entity certificate has no CKA_LABEL, then the alias is derived from the CKA_ID. If the CKA_ID can be determined to consist exclusively of printable characters, then a String alias is created by decoding the CKA_ID bytes using the UTF-8 charset. Otherwise, a hex String alias is created from the CKA_ID bytes ("0xFFFF...", for example).

If multiple certificates share the same CKA_LABEL, then the alias is derived from the CKA_LABEL plus the end entity certificate issuer and serial number (`"MyCert/CN=foobar/1234"`, for example).

4. Each certificate not part of a private key entry (as the end entity certificate) is checked whether it is trusted. If the CKA_TRUSTED attribute is true, then a KeyStore trusted certificate entry is created with the CKA_LABEL value as the KeyStore alias. If the certificate has no CKA_LABEL, or if multiple certificates share the same CKA_LABEL, then the alias is derived as described previously.
   If the CKA_TRUSTED attribute is not supported then no trusted certificate entries are created.
5. Any private key or certificate object not part of a private key entry or trusted certificate entry is ignored.
6. A search for all secret key objects on the token is performed by calling `C_FindObjects[Init|Final]`. The search template includes the following attributes:
   - CKA_TOKEN = true
   - CKA_CLASS = CKO_SECRET_KEY

   A KeyStore secret key entry is created for each secret key object, with the CKA_LABEL value as the KeyStore alias. Each secret key object must have a unique CKA_LABEL.

Write Access

To create new KeyStore entries on a PKCS#11 token to KeyStore entries, the SunPKCS11 provider's KeyStore implementation performs the following operations.

1. When creating a KeyStore entry (during KeyStore.setEntry, for example), C_CreateObject is called with `CKA_TOKEN=true` to create token objects for the respective entry contents. Private key objects are stored with `CKA_PRIVATE=true`. The KeyStore alias (UTF8-encoded) is set as the CKA_ID for both the private key and the corresponding end entity certificate. The KeyStore alias is also set as the CKA_LABEL for the end entity certificate object.

   Each certificate in a private key entry's chain is also stored. The CKA_LABEL is not set for CA certificates. If a CA certificate is already in the token, a duplicate is not stored.

   Secret key objects are stored with `CKA_PRIVATE=true`. The KeyStore alias is set as the CKA_LABEL.
2. If an attempt is made to convert a session object to a token object (for example, if KeyStore.setEntry is called and the private key object in the specified entry is a session object), then C_CopyObject is called with `CKA_TOKEN=true`.
3. If multiple certificates in the token are found to share the same CKA_LABEL, then the write capabilities to the token are disabled.
4. Since the PKCS#11 specification does not allow regular applications to set `CKA_TRUSTED=true` (only token initialization applications may do so), trusted certificate entries can not be created.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Miscellaneous

In addition to the searches listed previously, the following searches may be used by the SunPKCS11 provider's KeyStore implementation to perform internal functions. Specifically, `C_FindObjects[Init|Final]` may be called with any of the following attribute templates:

- ```
  CKA_TOKEN    true
  CKA_CLASS    CKO_CERTIFICATE
  CKA_SUBJECT  [subject DN]
  ```

- ```
  CKA_TOKEN    true
  CKA_CLASS    CKO_SECRET_KEY
  CKA_LABEL    [label]
  ```

- ```
  CKA_TOKEN    true
  CKA_CLASS    CKO_CERTIFICATE or CKO_PRIVATE_KEY
  CKA_ID       [cka_id]
  ```

Example Provider

The following is an example of a simple provider that demonstrates features of the `Provider` class.

```
package com.foo;

import java.io.*;
import java.lang.reflect.*;
import java.security.*;
import javax.crypto.*;

/**
 * Example provider that demonstrates some Provider class features.
 *
 *  . Implement multiple different algorithms in a single class.
 *    Previously each algorithm needed to be implemented in a separate class
 *    (e.g. one for SHA-256, one for SHA-384, etc.)
 *
 *  . Multiple concurrent instances of the provider frontend class each
 *    associated with a different backend.
 *
 *  . It uses "unextractable" keys and lets the framework know which key
 *    objects it can and cannot support
 *
 * Note that this is only a simple example provider designed to demonstrate
 * several of the new features. It is not explicitly designed for efficiency.
 */
public final class ExampleProvider extends Provider {

    // Reference to the crypto backend that implements all the algorithms.
    final CryptoBackend cryptoBackend;

    public ExampleProvider(String name, CryptoBackend cryptoBackend) {
        super(name, 1.0, "JCA/JCE provider for " + name);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.
```

```
        this.cryptoBackend = cryptoBackend;
        // register the algorithms we support (SHA-256, SHA-384, and AES)
        putService(new MyService
            (this, "MessageDigest", "SHA-256",
"com.foo.ExampleProvider$MyMessageDigest"));
        putService(new MyService
            (this, "MessageDigest", "SHA-384",
"com.foo.ExampleProvider$MyMessageDigest"));
        putService(new MyCipherService
            (this, "Cipher", "AES", "com.foo.ExampleProvider$MyCipher"));
    }

    // The API of our fictitious crypto backend.
    static abstract class CryptoBackend {
        abstract byte[] digest(String algorithm, byte[] data);
        abstract byte[] encrypt(String algorithm, KeyHandle key, byte[] data);
        abstract byte[] decrypt(String algorithm, KeyHandle key, byte[] data);
        abstract KeyHandle createKey(String algorithm, byte[] keyData);
    }

    // The shell of the representation the crypto backend uses for keys.
    private static final class KeyHandle {
        // fill in code
    }

    // We have our own ServiceDescription implementation that overrides
newInstance()
    // that calls the (Provider, String) constructor instead of the no-args
constructor.
    private static class MyService extends Service {

        private static final Class[] paramTypes = {Provider.class,
String.class};

        MyService(Provider provider, String type, String algorithm,
                String className) {
            super(provider, type, algorithm, className, null, null);
        }

        public Object newInstance(Object param) throws
NoSuchAlgorithmException {
            try {
                // Get the Class object for the implementation class.
                Class clazz;
                Provider provider = getProvider();
                ClassLoader loader = provider.getClass().getClassLoader();
                if (loader == null) {
                    clazz = Class.forName(getClassName());
                } else {
                    clazz = loader.loadClass(getClassName());
                }
                // Fetch the (Provider, String) constructor.
                Constructor cons = clazz.getConstructor(paramTypes);
                // Invoke constructor and return the SPI object.
                Object obj = cons.newInstance(new Object[] {provider,
getAlgorithm()});
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                return obj;
            } catch (Exception e) {
                throw new NoSuchAlgorithmException("Could not instantiate
service", e);
            }
        }
    }

    // Custom ServiceDescription class for Cipher objects. See
supportsParameter().
    private static class MyCipherService extends MyService {
        MyCipherService(Provider provider, String type, String algorithm,
                String className) {
            super(provider, type, algorithm, className);
        }
        // We override supportsParameter() to let the framework know which
        // keys we can support. We support instances of MySecretKey, if they
        // are stored in our provider backend, plus SecretKeys with a RAW
encoding.
        public boolean supportsParameter(Object obj) {
            if (obj instanceof SecretKey == false) {
                return false;
            }
            SecretKey key = (SecretKey)obj;
            if (key.getAlgorithm().equals(getAlgorithm()) == false) {
                return false;
            }
            if (key instanceof MySecretKey) {
                MySecretKey myKey = (MySecretKey)key;
                return myKey.provider == getProvider();
            } else {
                return "RAW".equals(key.getFormat());
            }
        }
    }

    // Our generic MessageDigest implementation. It implements all digest
    // algorithms in a single class. We only implement the bare minimum
    // of MessageDigestSpi methods.
    private static final class MyMessageDigest extends MessageDigestSpi {
        private final ExampleProvider provider;
        private final String algorithm;
        private ByteArrayOutputStream buffer;
        MyMessageDigest(Provider provider, String algorithm) {
            super();
            this.provider = (ExampleProvider)provider;
            this.algorithm = algorithm;
            engineReset();
        }
        protected void engineReset() {
            buffer = new ByteArrayOutputStream();
        }
        protected void engineUpdate(byte b) {
            buffer.write(b);
        }
        protected void engineUpdate(byte[] b, int ofs, int len) {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                buffer.write(b, ofs, len);
            }
            protected byte[] engineDigest() {
                byte[] data = buffer.toByteArray();
                byte[] digest = provider.cryptoBackend.digest(algorithm, data);
                engineReset();
                return digest;
            }
        }

        // our generic Cipher implementation, only partially complete. It
    implements
        // all cipher algorithms in a single class. We implement only as many of
    the
        // CipherSpi methods as required to show how it could work
        private static abstract class MyCipher extends CipherSpi {
            private final ExampleProvider provider;
            private final String algorithm;
            private int opmode;
            private MySecretKey myKey;
            private ByteArrayOutputStream buffer;
            MyCipher(Provider provider, String algorithm) {
                super();
                this.provider = (ExampleProvider)provider;
                this.algorithm = algorithm;
            }
            protected void engineInit(int opmode, Key key, SecureRandom random)
                    throws InvalidKeyException {
                this.opmode = opmode;
                myKey = MySecretKey.getKey(provider, algorithm, key);
                if (myKey == null) {
                    throw new InvalidKeyException();
                }
                buffer = new ByteArrayOutputStream();
            }
            protected byte[] engineUpdate(byte[] b, int ofs, int len) {
                buffer.write(b, ofs, len);
                return new byte[0];
            }
            protected int engineUpdate(byte[] b, int ofs, int len, byte[] out,
    int outOfs) {
                buffer.write(b, ofs, len);
                return 0;
            }
            protected byte[] engineDoFinal(byte[] b, int ofs, int len) {
                buffer.write(b, ofs, len);
                byte[] in = buffer.toByteArray();
                byte[] out;
                if (opmode == Cipher.ENCRYPT_MODE) {
                    out = provider.cryptoBackend.encrypt(algorithm, myKey.handle,
    in);
                } else {
                    out = provider.cryptoBackend.decrypt(algorithm, myKey.handle,
    in);
                }
                buffer = new ByteArrayOutputStream();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                return out;
            }
            // code for remaining CipherSpi methods goes here
        }

        // our SecretKey implementation. All our keys are stored in our crypto
        // backend, we only have an opaque handle available. There is no
        // encoded form of these keys.
        private static final class MySecretKey implements SecretKey {

            final String algorithm;
            final Provider provider;
            final KeyHandle handle;

            MySecretKey(Provider provider, String algorithm, KeyHandle handle) {
                super();
                this.provider = provider;
                this.algorithm = algorithm;
                this.handle = handle;
            }
            public String getAlgorithm() {
                return algorithm;
            }
            public String getFormat() {
                return null; // this key has no encoded form
            }
            public byte[] getEncoded() {
                return null; // this key has no encoded form
            }
            // Convert the given key to a key of the specified provider, if
possible
            static MySecretKey getKey(ExampleProvider provider, String algorithm,
Key key) {
                if (key instanceof SecretKey == false) {
                    return null;
                }
                // algorithm name must match
                if (!key.getAlgorithm().equals(algorithm)) {
                    return null;
                }
                // if key is already an instance of MySecretKey and is stored
                // on this provider, return it right away
                if (key instanceof MySecretKey) {
                    MySecretKey myKey = (MySecretKey)key;
                    if (myKey.provider == provider) {
                        return myKey;
                    }
                }
                // otherwise, if the input key has a RAW encoding, convert it
                if (!"RAW".equals(key.getFormat())) {
                    return null;
                }
                byte[] encoded = key.getEncoded();
                KeyHandle handle = provider.cryptoBackend.createKey(algorithm,
encoded);
                return new MySecretKey(provider, algorithm, handle);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                }
            }
        }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

6

Java Authentication and Authorization Service (JAAS)

> **Note**
>
> JAAS is linked with the `AccessControlContext` class and several methods from the `Subject` class, which are deprecated for removal. See Migrating from the Deprecated for Removal Subject.getSubject and Subject.doAs Methods to the Subject.current and Subject.callAs Methods for more information.

Java Authentication and Authorization Service (JAAS) Reference Guide describes Java Authentication and Authorization Service (JAAS), which enables you to authenticate users and securely determine who is currently executing Java code.

JAAS Authentication Tutorial describes how to use JAAS for the authentication of users: to reliably and securely determine who is currently running.

Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide shows you how to implement the `LoginModule` interface, which you plug into an application to provide a particular type of authentication.

Java Authentication and Authorization Service (JAAS) Reference Guide

You can use JAAS for the authentication of users, to reliably and securely determine who is currently executing Java code.

JAAS implements a Java version of the standard Pluggable Authentication Module (PAM) framework.

JAAS authentication is performed in a pluggable fashion. This permits applications to remain independent from underlying authentication technologies. New or updated authentication technologies can be plugged under an application without requiring modifications to the application itself. Applications enable the authentication process by instantiating a `LoginContext` object, which in turn references a `Configuration` to determine the authentication technology or technologies, or `LoginModule`(s), to be used in performing the authentication. Typical `LoginModules` may prompt for and verify a user name and password. Others may read and verify a voice or fingerprint sample.

The user or service executing the code is represented by a `Subject` object after it has been authenticated. The `Subject` is updated by a `LoginModule` with relevant `Principals` and credentials if authentication succeeds. An application or library can use the credentials stored in the `Subject` to make subsequent authorization decisions.

Topics

- Who Should Read This Document

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Related Documentation
- Core Classes and Interfaces
- JAAS Authorization
- Appendix A: JAAS Settings in the java.security Security Properties File
- Appendix B: JAAS Login Configuration File

Who Should Read This Document

This document is intended for experienced developers who require the ability to design applications constrained by a `Subject`-based security model. It is also intended to be read by `LoginModule` developers (developers implementing an authentication technology) prior to reading the Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide.

You may wish to first read JAAS Authentication Tutorial to get an overview of how to use JAAS and to see sample code in action, and then return to this document for further information.

Related Documentation

A supplement to this guide is the JAAS LoginModule Developer's Guide, intended for experienced programmers who require the ability to write a `LoginModule` implementing an authentication technology.

The JAAS Authentication Tutorial can be run by everyone.

JAAS Authentication is a similar tutorial that demonstrates the use of a Kerberos LoginModule. It requires a Kerberos installation. It's part of Introduction to JAAS and Java GSS-API Tutorials that utilize Kerberos as the underlying technology for authentication and secure communication.

Core Classes and Interfaces

The JAAS-related core classes and interfaces can be broken into two categories: Common and Authentication.

Topics

- Common Classes
- Authentication Classes and Interfaces

Common Classes

Common classes are those shared by both the JAAS authentication and authorization components.

The key JAAS class is `javax.security.auth.Subject`, which represents a grouping of related information for a single entity such as a person. It encompasses the entity's Principals, public credentials, and private credentials.

Note that the `java.security.Principal` interface is used to represent a Principal. Also note that a credential, as defined by JAAS, may be any Object.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Topics

- Subject
- Principals
- Credentials

Subject

To authorize access to resources, applications first need to authenticate the source of the request. The JAAS framework defines the term *subject* to represent the source of a request. A subject may be any entity, such as a person or a service. Once the subject is authenticated, a `javax.security.auth.Subject` is populated with associated identities, or Principals. A `Subject` may have many `Principals`. For example, a person may have a name `Principal` ("John Doe") and a SSN `Principal` ("123-45-6789"), which distinguish it from other subjects.

A `Subject` may also own security-related attributes, which are referred to as *credentials*; see the section Credentials. Sensitive credentials that require special protection, such as private cryptographic keys, are stored within a private credential `Set`. Credentials intended to be shared, such as public key certificates, are stored within a public credential `Set`.

Subjects are created using these constructors:

```
public Subject();
public Subject(boolean readOnly, Set principals,
               Set pubCredentials, Set privCredentials);
```

The first constructor creates a `Subject` with empty (non-null) `Sets` of `Principals` and credentials. The second constructor creates a `Subject` with the specified `Sets` of `Principals` and credentials. It also has a boolean argument which can be used to make the `Subject` read-only. In a read-only `Subject`, the `Principal` and credential `Sets` are immutable.

An application writer does not have to instantiate a `Subject`. If the application instantiates a `LoginContext` and does not pass a `Subject` to the `LoginContext` constructor, the `LoginContext` instantiates a new empty `Subject`. See the LoginContext section.

If a `Subject` was not instantiated to be in a read-only state, it can be set read-only by calling the following method:

```
public void setReadOnly();
```

Once in a read-only state, any attempt to add or remove `Principals` or credentials will result in an `IllegalStateException` being thrown. The following method may be called to test a `Subject`'s read-only state:

```
public boolean isReadOnly();
```

To retrieve the `Principals` associated with a `Subject`, two methods are available:

```
public Set getPrincipals();
public Set getPrincipals(Class c);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The first method returns all `Principals` contained in the `Subject`, while the second method only returns those `Principals` that are an instance of the specified `Class c`, or an instance of a subclass of `Class c`. An empty set will be returned if the `Subject` does not have any associated `Principals`.

To retrieve the public credentials associated with a `Subject`, these methods are available:

```
public Set getPublicCredentials();
public Set getPublicCredentials(Class c);
```

The behavior of these methods is similar to that for the `getPrincipals` methods, except in this case the public credentials are being obtained.

To access private credentials associated with a `Subject`, the following methods are available:

```
public Set getPrivateCredentials();
public Set getPrivateCredentials(Class c);
```

The behavior of these methods is similar to that for the `getPrincipals` and `getPublicCredentials` methods.

To modify or operate upon a `Subject`'s `Principal Set`, public credential `Set`, or private credential `Set`, callers use the methods defined in the `java.util.Set` class. The following example demonstrates this:

```
Subject subject;
Principal principal;
Object credential;
// ...

// add a Principal and credential to the Subject
subject.getPrincipals().add(principal);
subject.getPublicCredentials().add(credential);
```

> **Note**
>
> Only the sets returned through the `getPrincipals()`, `getPublicCredentials()`, and `getPrivateCredentials()` methods with no arguments are backed by the `Subject`'s respective internal sets. Therefore any modification to the returned set affects the internal sets as well. The sets returned through the `getPrincipals(Class c)`, `getPublicCredentials(Class c)`, and `getPrivateCredentials(Class c)` methods are not backed by the `Subject`'s respective internal sets. A new set is created and returned for each such method invocation. Modifications to these sets will not affect the `Subject`'s internal sets.

The following method enables you to perform an action as a specified `Subject`:

```
public static <T> T callAs(Subject subject, Callable<T> action)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

When you perform an action with the `callAs` method, the `Subject` is bound to the period of execution. The following method returns the `Subject` associated with the period of execution of the current thread or `null` if no `Subject` has been set:

```
public static Subject current();
```

See The callAs and current Methods for Performing an Action as a Particular Subject for more information.

The `Subject` class also includes the following methods inherited from `java.lang.Object`.

```
public boolean equals(Object o);
public String toString();
public int hashCode();
```

The callAs and current Methods for Performing an Action as a Particular Subject

Call the following static method to perform an action as a particular `Subject`:

```
public static <T> T callAs(Subject subject, Callable<T> action)
    throws CompletionException
```

The example SampleAcn.java from JAAS Authentication Tutorial calls `Subject.callAs(Subject, Callable<T>)` as follows:

```
Subject.callAs(mySubject, anotherAction);
```

In this example, the argument `anotherAction` accesses the `Subject` `mySubject` with the method `Subject.current()`:

```
Callable<Void> anotherAction = () -> {

    // Retrieve the current subject
    Subject s = Subject.current();
    System.out.println("\nCurrent subject: " + s);

    // Add a new Principal to the current subject
    Random r = new Random();
    String pn = Integer.toString(r.nextInt());
    Principal p = new sample.principal.SamplePrincipal(pn);

    System.out.println("\nAdding principal " + pn);
    s.getPrincipals().add(p);

    // List the current subject's Principals
    System.out.println("\nAuthenticated user has the following Principals:");
    Iterator pi = s.getPrincipals().iterator();

    while (pi.hasNext()) {
        Principal nextp = (Principal)pi.next();
        System.out.println("\t" + nextp.toString());
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    return null;
};
```

If a child thread starts and terminates within the execution of its parent's thread using structured concurrency, then the child thread can access the parent thread's current `Subject` with the `Subject.current()` method. The following example is from SampleAcn.java:

```
Callable<Void> addRandomPrincipal = () -> {

    Subject s = Subject.current();

    // Add a new Principal
    Random r = new Random();
    String pn = Integer.toString(r.nextInt());
    Principal p = new sample.principal.SamplePrincipal(pn);

    System.out.println("\nAdding principal " + pn);
    s.getPrincipals().add(p);
    return null;
};

Callable<Void> structuredAction = () -> {
    try (var scope = new StructuredTaskScope<>()) {
        scope.fork(addRandomPrincipal);
        scope.fork(addRandomPrincipal);
        scope.fork(addRandomPrincipal);

        scope.join();

        Subject s = Subject.current();
        System.out.println("\nCurrent subject: " + s);

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return null;
};

Subject.callAs(mySubject, structuredAction);
```

The method `StructuredTaskScope.fork(Callable)` starts a new thread in the task scope `scope`. This method returns a `StructuredTaskScope.Subtask` or a forked subtask. The example creates three forked subtasks. Each subtask obtains the parent thread's current `Subject` with the `Subject.current()` method, then adds a new principal to the `Subject` The method `StructuredTaskScope.join()` waits for the subtasks in `scope` to finish before continuing. The example then prints the `Subject`, which now contains three additional principals.

See Running the JAAS Authentication Tutorial Code to view the output from this example.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The class `StructuredTaskScope` is part of the structured concurrency API, which is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options, such as `--enable-preview -source 27`. See Preview Language and VM Features in *Java Platform, Standard Edition Java Language Updates* and Structured Concurrency in *Java Platform, Standard Edition Core Libraries*.

Principals

As mentioned previously, once a `Subject` is authenticated, it is populated with associated identities, or `Principals`. A `Subject` may have many `Principals`. For example, a person may have a name `Principal` ("John Doe") and an SSN `Principal` ("123-45-6789"), which distinguish it from other `Subjects`. A `Principal` must implement the `java.security.Principal` and `java.io.Serializable` interfaces. See Subject for information about ways to update the `Principals` associated with a `Subject`.

Credentials

In addition to associated Principals, a Subject may own security-related attributes, which are referred to as credentials. A credential may contain information used to authenticate the subject to new services. Such credentials include passwords, Kerberos tickets, and public key certificates. Credentials might also contain data that simply enables the subject to perform certain activities. Cryptographic keys, for example, represent credentials that enable the subject to sign or encrypt data. Public and private credential classes are not part of the core JAAS class library. Any class, therefore, can represent a credential.

Public and private credential classes are not part of the core JAAS class library. Developers, however, may elect to have their credential classes implement two interfaces related to credentials: `Refreshable` and `Destroyable`.

Topics

- Refreshable
- Destroyable

Refreshable

The `javax.security.auth.Refreshable` **interface** provides the capability for a credential to refresh itself. For example, a credential with a particular time-restricted lifespan may implement this interface to allow callers to refresh the time period for which it is valid. The interface has two abstract methods:

- `boolean isCurrent()`: Determines whether the credential is current or valid
- `void refresh() throws RefreshFailedException`: Updates or extends the validity of the credential

Destroyable

The `javax.security.auth.Destroyable` **interface** provides the capability of destroying the contents within a credential. The interface has two abstract methods:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `boolean isDestroyed()`: Determines whether the credential has been destroyed
- `void destroy() throws DestroyFailedException`: Destroys and clears the information associated with this credential

Authentication Classes and Interfaces

Authentication represents the process by which the identity of a subject is verified, and must be performed in a secure fashion; otherwise a perpetrator may impersonate others to gain access to a system. Authentication typically involves the subject demonstrating some form of evidence to prove its identity. Such evidence may be information only the subject would likely know or have (such as a password or fingerprint), or it may be information only the subject could produce (such as signed data using a private key).

To authenticate a subject (user or service), the following steps are performed:

1. An application instantiates a `LoginContext`.
2. The `LoginContext` consults a `Configuration` to load all of the `LoginModules` configured for that application.
3. The application invokes the `LoginContext`'s `login` method.
4. The `login` method invokes all of the loaded `LoginModules`. Each `LoginModule` attempts to authenticate the subject. Upon success, `LoginModules` associate relevant `Principals` and credentials with a `Subject` object that represents the subject being authenticated.
5. The `LoginContext` returns the authentication status to the application.
6. If authentication succeeded, the application retrieves the `Subject` from the `LoginContext`.

The following sections describe the authentication classes.

Topics

- LoginContext
- LoginModule
- CallbackHandler
- Callback

LoginContext

The `javax.security.auth.login.LoginContext` class provides the basic methods used to authenticate subjects, and provides a way to develop an application independent of the underlying authentication technology. The `LoginContext` consults a `Configuration` to determine the authentication services, or `LoginModule(s)`, configured for a particular application. Therefore, different `LoginModules` can be plugged in under an application without requiring any modifications to the application itself.

`LoginContext` offers four constructors from which to choose:

```
public LoginContext(String name) throws LoginException
public LoginContext(String name, Subject subject) throws LoginException
public LoginContext(String name, CallbackHandler callbackHandler)
           throws LoginException
public LoginContext(String name, Subject subject,
           CallbackHandler callbackHandler) throws LoginException
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

All of the constructors share a common parameter: `name`. This argument is used by the `LoginContext` as an index into the login `Configuration` to determine which `LoginModules` are configured for the application instantiating the `LoginContext`. Constructors that do not take a `Subject` as an input parameter instantiate a new `Subject`. Null inputs are disallowed for all constructors.

See CallbackHandler for information on what a `CallbackHandler` is and when you may need one.

Actual authentication occurs with a call to the following method:

```
public void login() throws LoginException
```

When `login` is invoked, all of the configured `LoginModules` are invoked to perform the authentication. If the authentication succeeded, the `Subject` (which may now hold `Principals`, public credentials, and private credentials) can be retrieved by using the following method:

```
public Subject getSubject()
```

To logout a `Subject` and remove its authenticated `Principals` and credentials, the following method is provided:

```
public void logout() throws LoginException
```

The following code sample demonstrates the calls necessary to authenticate and logout a `Subject`:

```
// let the LoginContext instantiate a new Subject
LoginContext lc = new LoginContext("entryFoo");
try {
    // authenticate the Subject
    lc.login();
    System.out.println("authentication successful");

    // get the authenticated Subject
    Subject subject = lc.getSubject();

    // ...

    // all finished -- logout
    lc.logout();
} catch (LoginException le) {
     System.err.println("authentication unsuccessful: " +
         le.getMessage());
}
```

LoginModule

The `LoginModule` **interface** gives developers the ability to implement different kinds of authentication technologies that can be plugged in under an application. For example, one type of `LoginModule` may perform a user name/password-based form of authentication. Other `LoginModules` may interface to hardware devices such as smart cards or biometric devices.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> If you are an application writer, you do not need to understand the workings of `LoginModules`. All you have to know is how to write your application and specify configuration information (such as in a login configuration file) such that the application will be able to utilize the LoginModule specified by the configuration to authenticate the user.

If, on the other hand, you are a programmer who wishes to write a LoginModule implementing an authentication technology, see the Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide for detailed step-by-step instructions.

CallbackHandler

In some cases a `LoginModule` must communicate with the user to obtain authentication information. `LoginModules` use a `javax.security.auth.callback.CallbackHandler` for this purpose. Applications implement the `CallbackHandler` **interface** and pass it to the `LoginContext`, which forwards it directly to the underlying `LoginModules`. A `LoginModule` uses the `CallbackHandler` both to gather input from users (such as a password or smart card pin number) or to supply information to users (such as status information). By allowing the application to specify the `CallbackHandler`, underlying `LoginModules` can remain independent of the different ways applications interact with users. For example, the implementation of a `CallbackHandler` for a GUI application might display a window to solicit input from a user. On the other hand, the implementation of a `CallbackHandler` for a non-GUI tool might simply prompt the user for input directly from the command line.`CallbackHandler` is an interface with one method to implement:

```
void handle(Callback[] callbacks)
    throws java.io.IOException, UnsupportedCallbackException;
```

The `LoginModule` passes the `CallbackHandler handle` method an array of appropriate `Callbacks`, for example a `NameCallback` for the user name and a `PasswordCallback` for the password, and the `CallbackHandler` performs the requested user interaction and sets appropriate values in the `Callbacks`. For example, to process a `NameCallback`, the `CallbackHandler` may prompt for a name, retrieve the value from the user, and call the `NameCallback`'s `setName` method to store the name.

The `CallbackHandler` documentation has a lengthy example not included in this document that readers may want to examine.

Callback

The `javax.security.auth.callback` package contains the Callback **interface** as well as several implementations. LoginModules may pass an array of Callbacks directly to the handle method of a CallbackHandler.

Please consult the various Callback APIs for more information on their use.

JAAS Authorization

Before the Security Manager was permanently disabled in JDK 24, the JAAS authorization component ensured users had the access control rights (or permissions) required to do the actions performed. Once the user or service executing the code was authenticated, the JAAS

Copyright © 1993, 2026, Oracle and/or its affiliates.

authorization component worked in conjunction with the core Java SE access control model and the Security Manager to protect access to sensitive resources.

This authorization mechanism is no longer supported because it's dependent on policy files and permissions, both of which have been removed or disabled now that the Security Manager is permanently disabled. Consequently, application code is responsible for performing authorization. This can be based on the principals and credentials that are stored inside a `Subject` after authentication.

For example, consider the example described in the JAAS Authentication Tutorial. The SampleLoginModule.java class, which implements the `LoginModule` interface, authenticates the user with a password. After it has authenticated the user, information (such as principals and credentials) can be extracted from the `Subject` and used for subsequent authorization decisions. The SampleAcn.java class runs the `Callable` `anotherAction` with the authenticated `Subject` with the `Subject.callAs` method:

```
Subject.callAs(mySubject, anotherAction);
```

Let's assume that the current `Subject` has one principal. The following example adds code to an excerpt from `SampleAcn.java` that retrieves this principal and grants access based on the principal's identity:

```
Callable<Void> anotherAction = () -> {

    Subject s = Subject.current();
    System.out.println("\nCurrent subject: " + s);

    Optional<Principal> principal = s.getPrincipals().stream().findFirst();

    // Authorize principal
    if (principal.isPresent() &&
        principal.get().getName().equals("Duke")) {
        // Grant access (code not shown)
        // ...
    }
}
```

Appendix A: JAAS Settings in the java.security Security Properties File

A number of JAAS-related settings can be configured in the `java.security` master Security Properties file, which is located in the `conf/security` directory of the JDK.

JAAS adds two new security properties to `java.security`:

- `login.configuration.provider`
- `login.config.url.`*`n`*

The following example demonstrates how to configure these properties. In this example, we leave the values provided in the default `java.security` file for the `login.configuration.provider` Security Property. The default `java.security` file also lists a value for the `login.config.url.`*`n`* Security Property, but it is commented out. In the following example, it is not commented.

```
#
# Class to instantiate as the javax.security.auth.login.Configuration
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
# provider.
#
login.configuration.provider=sun.security.provider.ConfigFile

#
# Default login configuration file
#
login.config.url.1=file:${user.home}/.java.login.config
```

> **Note**
>
> Modifications made to this file may be overwritten by subsequent JDK updates. However, an alternate `java.security` properties file may be specified from the command line via the system property `java.security.properties=<URL>`. This properties file appends to the system properties file. If both properties files specify values for the same key, the value from command-line properties file is selected, as it is the last one loaded.
>
> Also, specifying `java.security.properties==<URL>` (using two equals signs), then that properties file will completely override the system properties file.
>
> To disable the ability to specify an additional properties file from the command line, set the key `security.overridePropertiesFile` to `false` in the system properties file. It is set to `true` by default.

Topics

- Login Configuration Provider
- Login Configuration URLs

Login Configuration Provider

The default JAAS login configuration implementation provided by Oracle gets its configuration information from files and expects the information to be provided in a specific format shown in the tutorials.

The default JAAS login configuration implementation can be replaced by specifying the alternative provider class implementation in the `login.configuration.provider` property.

For example:

```
login.configuration.provider=com.foo.Config
```

If the Security property `login.configuration.provider` is not found, or is left unspecified, then it is set to the default value:

```
login.configuration.provider=com.sun.security.auth.login.ConfigFile
```

Note that there is no means to dynamically set the login configuration provider from the command line.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Login Configuration URLs

If you are using a login configuration implementation that expects the configuration information to be specified in files (as does the default implementation from Oracle), the location of the login configuration file(s) can be statically set by specifying their respective URLs in the `login.config.url.n` property. '*n*' is a consecutively numbered integer starting with 1. If multiple configuration files are specified (if *n* >= 2), they will be read and unioned into one single configuration.

For example:

```
login.config.url.1=file:C:/config/.java.login.config
login.config.url.2=file:C:/users/foo/.foo.login.config
```

If the location of the configuration files is not set in the `java.security` properties file, and also is not specified dynamically from the command line (via the `-Djava.security.auth.login.config` option), JAAS attempts to load a default configuration from

```
file:${user.home}/.java.login.config
```

Appendix B: JAAS Login Configuration File

JAAS authentication is performed in a pluggable fashion, so Java applications can remain independent from underlying authentication technologies. Configuration information such as the desired authentication technology is specified at runtime. The source of the configuration information (for example, a file or a database) is up to the current `javax.security.auth.login.Configuration` implementation. The default `Configuration` implementation, `ConfigFile`, gets its configuration information from login configuration files. For details about the default login `Configuration` implementation provided with JAAS, see the `com.sun.security.auth.login.ConfigFile` class.

Topics

- Login Configuration File Structure and Contents
- Where to Specify Which Login Configuration File Should Be Used

Login Configuration File Structure and Contents

A login configuration file consists of one or more entries, each specifying which underlying authentication technology should be used for a particular application or applications. The structure of each entry is the following:

```
<name used by application to refer to this entry> {
    <LoginModule> <flag> <LoginModule options>;
    <optional additional LoginModules, flags and options>;
};
```

Thus, each login configuration file entry consists of a name followed by one or more `LoginModule`-specific entries, where each `LoginModule`-specific entry is terminated by a

Copyright © 1993, 2026, Oracle and/or its affiliates.

semicolon and the entire group of `LoginModule`-specific entries is enclosed in braces. Each configuration file entry is terminated by a semicolon.

Example 6-1 Login Configuration File for JAAS Authentication Tutorial

As an example, the login configuration file used for the JAAS Authentication Tutorial tutorial contains just one entry, which is

```
Sample {
   sample.module.SampleLoginModule required debug=true;
};
```

Here, the entry is named `Sample` and that is the name that the JAAS Authentication tutorial application (`SampleAcn.java`) uses to refer to this entry. The entry specifies that the `LoginModule` to be used to do the user authentication is the `SampleLoginModule` in the `sample.module` package and that this `SampleLoginModule` is required to "succeed" in order for authentication to be considered successful. The `SampleLoginModule` succeeds only if the name and password supplied by the user are the ones it expects (`testUser` and `testPassword`, respectively).

The **name** for an entry in a login configuration file is the name that applications use to refer to the entry when they instantiate a `LoginContext`, as described in JAAS Authentication Tutorial in the JAAS authentication tutorial. The name can be whatever name the application developer wishes to use. Here, the term "application" refers to whatever code does the JAAS login.

The specified `LoginModules` are used to control the authentication process. Authentication proceeds down the list in the exact order specified, as described in the `Configuration` class.

The subparts of each `LoginModule`-specific entry are the following:

- **LoginModule**: This specifies a class implementing the desired authentication technology. Specifically, the class must be a subclass of the `LoginModule` class, which is in the `javax.security.auth.spi` package. A typical `LoginModule` may prompt for and verify a user name and password, as is done by the `SampleLoginModule` (in the `sample.module` package) used for these tutorials. Any vendor can provide a LoginModule implementation that you can use. Some implementations are supplied with the JDK from Oracle. You can view the reference documentation for the various `LoginModules`, all in the `com.sun.security.auth` package:
  - `JndiLoginModule`
  - `KeyStoreLoginModule`
  - `Krb5LoginModule`
  - `NTLoginModule`
  - `UnixLoginModule`
- **flag**: The flag value indicates whether success of the preceding `LoginModule` is `required`, `requisite`, `sufficient`, or `optional`. If there is just one `LoginModule`-specific entry, as there is in our tutorials, then the flag for it should be "required". The options are described in more detail in the `Configuration` class.
- **LoginModule options**: If the specified `LoginModule` implementation has options that can be set, you specify any desired option values here. This is a space-separated list of values which are passed directly to the underlying `LoginModule`. Options are defined by the `LoginModule` itself, and control the behavior within it. For example, a `LoginModule` may define options to support debugging/testing capabilities.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The correct way to specify options in the configuration file is by using a name-value pairing, for example `debug=true`, where the option name (in this case, `debug`) and value (in this case, `true`) should be separated by an equals symbol.

Example 6-2 Login Configuration File Demonstrating required, sufficient, requisite, and optional Flags

The following is a sample login configuration file that demonstrates the `required`, `sufficient`, `requisite`, and `optional` flags. See the `Configuration` class for more information about these flags.

```
Login1 {
       sample.SampleLoginModule required debug=true;
    };

    Login2 {
       sample.SampleLoginModule required;
       com.sun.security.auth.module.NTLoginModule sufficient;
       com.foo.SmartCard requisite debug=true;
       com.foo.Kerberos optional debug=true;
    };
```

The application **Login1** only has one configured `LoginModule`, `SampleLoginModule`. Therefore, an attempt by **Login1** to authenticate a subject (user or service) will be successful if and only if the `SampleLoginModule` succeeds.

The authentication logic for the application **Login2** is easier to explain with the following table:

Table 6-1 Login2 Authentication Status

Module Class	Flag	Authentication Attempt 1	Authentication Attempt 2	Authentication Attempt 3	Authentication Attempt 4	Authentication Attempt 5	Authentication Attempt 6	Authentication Attempt 7	Authentication Attempt 8
`SampleLoginModule`	required	pass	pass	pass	pass	fail	fail	fail	fail
`NTLoginModule`	sufficient	pass	fail	fail	fail	pass	fail	fail	fail
SmartCard	requisite	*	pass	pass	fail	*	pass	pass	fail
Kerberos	optional	*	pass	fail	*	*	pass	fail	*
Overall Authentication	not applicable	pass	pass	pass	fail	fail	fail	fail	fail

* = trivial value due to control returning to the application because a previous *requisite* module failed or a previous *sufficient* module succeeded.

Where to Specify Which Login Configuration File Should Be Used

The configuration file to be used can be specified in one of two ways:

1. On the command line.

Copyright © 1993, 2026, Oracle and/or its affiliates.

You can use a `-Djava.security.auth.login.config` interpreter command line argument to specify the login configuration file that should be used. We use this approach for all the tutorials. For example, we run our `SampleAcn` application in the JAAS Authentication Tutorial using the following command, which specifies that the configuration file is the `sample_jaas.config` file in the current directory:

```
java -Djava.security.auth.login.config==sample_jaas.config sample.SampleAcn
```

> **Note**
>
> If you use a single equals sign (=) with the `java.security.auth.login.config` system property (instead of a double equals sign (==)), then the configurations specified by both this system property and the `java.security` file are used.

2. In the Java Security Properties file.

   An alternate approach to specifying the location of the login configuration file is to indicate its URL as the value of a `login.config.url.`*n* property in the security properties file. The Security Properties file is the `java.security` file located in the `conf/security` directory of the JDK.

   Here, *n* indicates a consecutively-numbered integer starting with 1. Thus, if desired, you can specify more than one login configuration file by indicating one file's URL for the `login.config.url.1` property, a second file's URL for the `login.config.url.2` property, and so on. If more than one login configuration file is specified (that is, if *n* > 1), then the files are read and concatenated into a single configuration.

   Here is an example of what would need to be added to the security properties file in order to indicate the `sample_jaas.config` login configuration file used by this tutorial. This example assumes the file is in the `C:\AcnTest` directory on Windows:

   ```
   login.config.url.1=file:C:/AcnTest/sample_jaas.config
   ```

   (Note that URLs always use forward slashes, regardless of what operating system the user is running.)

JAAS Authentication Tutorial

This tutorial describes how to use JAAS for the authentication of users: to reliably and securely determine who is currently running.

JAAS authentication is performed in a pluggable fashion. This permits Java applications to remain independent from underlying authentication technologies. New or updated technologies can be plugged in without requiring modifications to the application itself. An implementation for a particular authentication technology to be used is determined at runtime. The implementation is specified in a login configuration file. The authentication technology used for this tutorial is very basic, just ensuring that the user specifies a particular name and password.

The rest of this tutorial consists of the following sections:

1. The Authentication Tutorial Code
2. The Login Configuration
3. Running the JAAS Authentication Tutorial Code

Copyright © 1993, 2026, Oracle and/or its affiliates.

If you want to first see the tutorial code in action, you can skip directly to Running the JAAS Authentication Tutorial Code and then go back to the other sections to learn about coding and configuration file details.

The Authentication Tutorial Code

The code for this tutorial consists of three files:

- `SampleAcn.java` contains the sample application class (`SampleAcn`) and another class used to handle user input (`MyCallbackHandler`). **The code in this file is the only code you need to understand for this tutorial. Your application will only indirectly use the other source files.**
- `SampleLoginModule.java` is the class specified by the tutorial's login configuration file, `sample_jaas.config`, described in The Login Configuration File for the JAAS Authentication Tutorial as the class implementing the desired underlying authentication. `SampleLoginModule`'s user authentication consists of simply verifying that the name and password specified by the user have specific values.
- `SamplePrincipal.java` is a sample class implementing the `java.security.Principal` interface. It is used by `SampleLoginModule`.

SampleAcn.java

Our authentication tutorial application code is contained in a single source file, `SampleAcn.java`. That file contains two classes:

- The SampleAcn Class
- The MyCallbackHandler Class

The SampleAcn Class

The `main` method of the `SampleAcn` class performs the authentication and then reports whether or not authentication succeeded.

The code for authenticating the user is very simple, consisting of just two steps:

1. Instantiating a LoginContext
2. Calling the LoginContext's login Method

> **Note**
>
> The `SampleAcn` class also demonstrates the following:
>
> - How to run code with a specified `Subject` with the `Subject.callAs(Subject, Callable<T>)` method
> - How to access the parent thread's current `Subject` from a child thread using structured concurrency.
>
> See The callAs and current Methods for Performing an Action as a Particular Subject for more information.

First the basic code is shown, followed by The Complete SampleAcn Class Code, complete with the import statement it requires and error handling.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Instantiating a LoginContext

In order to authenticate a user, you first need a `javax.security.auth.login.LoginContext`. Here is the basic way to instantiate a LoginContext:

```
import javax.security.auth.login.*;
// ...
LoginContext lc =
    new LoginContext(<config file entry name>,
           <CallbackHandler to be used for user interaction>);
```

and here is the specific way our tutorial code does the instantiation:

```
import javax.security.auth.login.*;
// ...
LoginContext lc =
    new LoginContext("Sample",
          new MyCallbackHandler());
```

The arguments are the following:

1. **The name of an entry in the JAAS login configuration file**
   This is the name for the `LoginContext` to use to look up an entry for this application in the JAAS login configuration file, described in The Login Configuration. Such an entry specifies the class(es) that implement the desired underlying authentication technology(ies). The class(es) must implement the `LoginModule` interface, which is in the `javax.security.auth.spi` package.

   In our sample code, we use the `SampleLoginModule` supplied with this tutorial. The `SampleLoginModule` performs authentication by ensuring that the user types a particular name and password.

   The entry in the login configuration file we use for this tutorial, `sample_jaas.config` (see The Login Configuration File for the JAAS Authentication Tutorial), has the name "Sample", so that is the name we specify as the first argument to the LoginContext constructor.

2. **A CallbackHandler instance**
   When a `LoginModule` needs to communicate with the user, for example to ask for a user name and password, it does not do so directly. That is because there are various ways of communicating with a user, and it is desirable for `LoginModules` to remain independent of the different types of user interaction. Rather, the `LoginModule` invokes a `javax.security.auth.callback.CallbackHandler` to perform the user interaction and obtain the requested information, such as the user name and password.

   An instance of the particular `CallbackHandler` to be used is specified as the second argument to the `LoginContext` constructor. The `LoginContext` forwards that instance to the underlying `LoginModule` (in our case `SampleLoginModule`). An application typically provides its own `CallbackHandler` implementation. A simple `CallbackHandler`, `TextCallbackHandler`, is provided in the `com.sun.security.auth.callback` package to output information to and read input from the command line. However, we instead demonstrate the more typical case of an application providing its own `CallbackHandler` implementation, described in The MyCallbackHandler Class.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Calling the LoginContext's login Method

Once we have a `LoginContext` `lc`, we can call its `login` method to carry out the authentication process:

```
lc.login();
```

The `LoginContext` instantiates a new empty `javax.security.auth.Subject` object (which represents the user or service being authenticated; see Subject). The `LoginContext` constructs the configured `LoginModule` (in our case `SampleLoginModule`) and initializes it with this new `Subject` and `MyCallbackHandler`.

The `LoginContext`'s `login` method then calls methods in the `SampleLoginModule` to perform the login and authentication. The `SampleLoginModule` will utilize the `MyCallbackHandler` to obtain the user name and password. Then the `SampleLoginModule` will check that the name and password are the ones it expects.

If authentication is successful, the `SampleLoginModule` populates the `Subject` with a `Principal` representing the user. The `Principal` the `SampleLoginModule` places in the `Subject` is an instance of `SamplePrincipal`, which is a sample class implementing the `java.security.Principal` interface.

The calling application can subsequently retrieve the authenticated `Subject` by calling the `LoginContext`'s `getSubject` method, although doing so is not necessary for this tutorial.

The Complete SampleAcn Class Code

Now that you have seen the basic code required to authenticate the user, we can put it all together into the full class in `SampleAcn.java`, which includes relevant import statements and error handling:

SampleAcn.java

```
package sample;

import java.io.*;
import java.util.*;
import javax.security.auth.login.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import java.security.Principal;
import java.util.concurrent.Callable;
import java.util.concurrent.StructuredTaskScope;

/**
 * <p>This sample application attempts to authenticate a user and reports
whether
 * or not the authentication was successful.</p>
 *
 * <p>It also demonstrates how to run code with a specified Subject with the
 * Subject.callAs(Subject, Callable<T? method and how to access the parent
 * thread's current Subject from a child thread using structured concurrency
and
 * the Subject.current() method.</p>
 */
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
public class SampleAcn {

   /**
    * <p>Attempt to authenticate the user.</p>
    *
    * @param args input arguments for this application. These are ignored.
    */

    public static void main(String[] args) {

        // Obtain a LoginContext, needed for authentication. Tell it
        // to use the LoginModule implementation specified by the
        // entry named "Sample" in the JAAS login configuration
        // file and to also use the specified CallbackHandler.

        LoginContext lc = null;
        try {
            lc = new LoginContext("Sample", new MyCallbackHandler());
        } catch (LoginException le) {
            System.err.println("Cannot create LoginContext. "
                + le.getMessage());
            System.exit(-1);
        }

        // The user has 3 attempts to authenticate successfully
        int i;
        for (i = 0; i < 3; i++) {
            try {

                // Attempt authentication
                lc.login();

                // If we return with no exception, authentication succeeded
                break;

            } catch (LoginException le) {

                System.err.println("Authentication failed:");
                System.err.println("  " + le.getMessage());
                try {
                    Thread.currentThread().sleep(3000);
                } catch (Exception e) {
                    // ignore
                }
            }
        }

        // did they fail three times?
        if (i == 3) {
            System.out.println("Sorry");
            System.exit(-1);
        }

        System.out.println("Authentication succeeded!");

        Subject mySubject = lc.getSubject();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        // Let's see what Principals we have
        Iterator principalIterator = mySubject.getPrincipals().iterator();
        System.out.println("Authenticated user has the following
Principals:");

        while (principalIterator.hasNext()) {
            Principal p = (Principal)principalIterator.next();
            System.out.println("\t" + p.toString());
        }

        System.out.println("User has " +
                           mySubject.getPublicCredentials().size() +
                           " Public Credential(s)");

        Callable<Void> anotherAction = () -> {

            // Retrieve the current subject
            Subject s = Subject.current();
            System.out.println("\nCurrent subject: " + s);

            // Add a new Principal to the current subject
            Random r = new Random();
            String pn = Integer.toString(r.nextInt());
            Principal p = new sample.principal.SamplePrincipal(pn);

            System.out.println("\nAdding principal " + pn);
            s.getPrincipals().add(p);

            // List the current subject's Principals
            System.out.println("\nAuthenticated user has the following
Principals:");
            Iterator pi = s.getPrincipals().iterator();

            while (pi.hasNext()) {
                Principal nextp = (Principal)pi.next();
                System.out.println("\t" + nextp.toString());
            }

            return null;
        };

        // Run the Callable anotherAction with the current subject

        Subject.callAs(mySubject, anotherAction);

        // The following demonstrates how to retrieve the parent thread's
        // current subject from a child thread with structured concurrency.

        Callable<Void> addRandomPrincipal = () -> {

            Subject s = Subject.current();

            // Add a new Principal
            Random r = new Random();
            String pn = Integer.toString(r.nextInt());
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                Principal p = new sample.principal.SamplePrincipal(pn);

                System.out.println("\nAdding principal " + pn);
                s.getPrincipals().add(p);
                return null;
            };

            Callable<Void> structuredAction = () -> {
                try (var scope = new StructuredTaskScope<>()) {
                    scope.fork(addRandomPrincipal);
                    scope.fork(addRandomPrincipal);
                    scope.fork(addRandomPrincipal);

                    scope.join();

                    Subject s = Subject.current();
                    System.out.println("\nCurrent subject: " + s);

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return null;
            };

            // Run the Callable anotherAction with the current subject

            Subject.callAs(mySubject, structuredAction);
        }
    }


    /**
     * <p>The application implements the CallbackHandler.</p>
     *
     * <p>This application is text-based. Therefore, it displays information
     * to the user using the OutputStreams System.out and System.err,
     * and gathers input from the user using the InputStream System.in.</p>
     */

    class MyCallbackHandler implements CallbackHandler {

        /**
         * <p>Invoke an array of Callbacks.</p>
         *
         * @param callbacks an array of <code>Callback</code> objects which
    contain
         *                  the information requested by an underlying security
         *                  service to be retrieved or displayed.
         *
         * @exception java.io.IOException if an input or output error occurs. <p>
         *
         * @exception UnsupportedCallbackException if the implementation of this
         *                  method does not support one or more of the Callbacks
         *                  specified in the <code>callbacks</code> parameter.
         */
        public void handle(Callback[] callbacks)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
throws IOException, UnsupportedCallbackException {

    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof TextOutputCallback) {

            // display the message according to the specified type
            TextOutputCallback toc = (TextOutputCallback)callbacks[i];
            switch (toc.getMessageType()) {
            case TextOutputCallback.INFORMATION:
                System.out.println(toc.getMessage());
                break;
            case TextOutputCallback.ERROR:
                System.out.println("ERROR: " + toc.getMessage());
                break;
            case TextOutputCallback.WARNING:
                System.out.println("WARNING: " + toc.getMessage());
                break;
            default:
                throw new IOException("Unsupported message type: " +
                                      toc.getMessageType());
            }

        } else if (callbacks[i] instanceof NameCallback) {

            // prompt the user for a username
            NameCallback nc = (NameCallback)callbacks[i];

            System.err.print(nc.getPrompt());
            System.err.flush();
            nc.setName((new BufferedReader
                    (new InputStreamReader(System.in))).readLine());

        } else if (callbacks[i] instanceof PasswordCallback) {

            // prompt the user for sensitive information
            PasswordCallback pc = (PasswordCallback)callbacks[i];
            System.err.print(pc.getPrompt());
            System.err.flush();
            pc.setPassword(readPassword(System.in));

        } else {
            throw new UnsupportedCallbackException
                    (callbacks[i], "Unrecognized Callback");
        }
    }
}

// Reads user password from given input stream.
private char[] readPassword(InputStream in) throws IOException {

    char[] lineBuffer;
    char[] buf;
    int i;

    buf = lineBuffer = new char[128];
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            int room = buf.length;
            int offset = 0;
            int c;

loop:       while (true) {
                switch (c = in.read()) {
                case -1:
                case '\n':
                    break loop;

                case '\r':
                    int c2 = in.read();
                    if ((c2 != '\n') && (c2 != -1)) {
                        if (!(in instanceof PushbackInputStream)) {
                            in = new PushbackInputStream(in);
                        }
                        ((PushbackInputStream)in).unread(c2);
                    } else
                        break loop;

                default:
                    if (--room < 0) {
                        buf = new char[offset + 128];
                        room = buf.length - offset - 1;
                        System.arraycopy(lineBuffer, 0, buf, 0, offset);
                        Arrays.fill(lineBuffer, ' ');
                        lineBuffer = buf;
                    }
                    buf[offset++] = (char) c;
                    break;
                }
            }

            if (offset == 0) {
                return null;
            }

            char[] ret = new char[offset];
            System.arraycopy(buf, 0, ret, 0, offset);
            Arrays.fill(buf, ' ');

            return ret;
        }
    }
```

The MyCallbackHandler Class

In some cases a `LoginModule` must communicate with the user to obtain authentication information. `LoginModules` use a `javax.security.auth.callback.CallbackHandler` for this purpose. An application can either use one of the sample implementations provided in the `com.sun.security.auth.callback` package or, more typically, write a `CallbackHandler` implementation. The application passes the `CallbackHandler` as an argument to the `LoginContext` instantiation. The `LoginContext` forwards the `CallbackHandler` directly to the underlying `LoginModules`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The tutorial sample code supplies its own `CallbackHandler` implementation, the `MyCallbackHandler` class in page 17.

`CallbackHandler` is an interface with one method to implement:

```
     void handle(Callback[] callbacks)
         throws java.io.IOException, UnsupportedCallbackException;
```

The `LoginModule` passes the `CallbackHandler` handle method an array of appropriate `javax.security.auth.callback.Callbacks`, for example a `NameCallback` for the user name and a `PasswordCallback` for the password, and the `CallbackHandler` performs the requested user interaction and sets appropriate values in the `Callbacks`.

The `MyCallbackHandler handle` method is structured as follows:

```
public void handle(Callback[] callbacks)
  throws IOException, UnsupportedCallbackException {

  for (int i = 0; i < callbacks.length; i++) {
    if (callbacks[i] instanceof TextOutputCallback) {

      // display a message according to a specified type
      . . .

    } else if (callbacks[i] instanceof NameCallback) {

      // prompt the user for a username
      . . .

    } else if (callbacks[i] instanceof PasswordCallback) {

      // prompt the user for a password
      . . .

    } else {
        throw new UnsupportedCallbackException
         (callbacks[i], "Unrecognized Callback");
    }
  }
}
```

A `CallbackHandler handle` method is passed an array of `Callback` instances, each of a particular type (`NameCallback`, `PasswordCallback`, etc.). It must handle each Callback, performing user interaction in a way that is appropriate for the executing application.

`MyCallbackHandler` handles three types of Callbacks: **`NameCallback`** to prompt the user for a user name, **`PasswordCallback`** to prompt for a password, and **`TextOutputCallback`** to report any error, warning, or other messages the `SampleLoginModule` wishes to send to the user.

The `handle` method handles a **`TextOutputCallback`** by extracting the message to be reported and then printing it to `System.out`, optionally preceded by additional wording that depends on the message type. The message to be reported is determined by calling the

Copyright © 1993, 2026, Oracle and/or its affiliates.

`TextOutputCallback`'s `getMessage` method and the type by calling its `getMessageType` method. Here is the code for handling a `TextOutputCallback`:

```
if (callbacks[i] instanceof TextOutputCallback) {

  // display the message according to the specified type
  TextOutputCallback toc = (TextOutputCallback)callbacks[i];
  switch (toc.getMessageType()) {
     case TextOutputCallback.INFORMATION:
        System.out.println(toc.getMessage());
        break;
     case TextOutputCallback.ERROR:
        System.out.println("ERROR: " + toc.getMessage());
        break;
     case TextOutputCallback.WARNING:
        System.out.println("WARNING: " + toc.getMessage());
        break;
     default:
        throw new IOException("Unsupported message type: " +
            toc.getMessageType());
   }
```

The `handle` method handles a **`NameCallback`** by prompting the user for a user name. It does this by printing the prompt to `System.err`. It then sets the name for use by the `SampleLoginModule` by calling the `NameCallback`'s `setName` method, passing it the name typed by the user:

```
} else if (callbacks[i] instanceof NameCallback) {

    // prompt the user for a username
    NameCallback nc = (NameCallback)callbacks[i];

    System.err.print(nc.getPrompt());
    System.err.flush();
    nc.setName((new BufferedReader
        (new InputStreamReader(System.in))).readLine());
```

Similarly, the `handle` method handles a **`PasswordCallback`** by printing a prompt to `System.err` to prompt the user for a password. It then sets the password for use by the `SampleLoginModule` by calling the `PasswordCallback`'s `setPassword` method, passing it the password typed by the user:

```
} else if (callbacks[i] instanceof PasswordCallback) {

    // prompt the user for sensitive information
    PasswordCallback pc = (PasswordCallback)callbacks[i];

    System.err.print(pc.getPrompt());
    System.err.flush();
    pc.setPassword(System.console().readPassword());
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

SampleLoginModule.java and SamplePrincipal.java

`SampleLoginModule.java` implements the `LoginModule` interface. `SampleLoginModule` is the class specified by the tutorial's login configuration file (see The Login Configuration File for the JAAS Authentication Tutorial) as the class implementing the desired underlying authentication. `SampleLoginModule`'s user authentication consists of simply verifying that the name and password specified by the user have specific values. This `SampleLoginModule` is specified by the tutorial's login configuration file as the `LoginModule` to use because (1) It performs a basic type of authentication suitable for any environment and thus is appropriate for a tutorial for all users, and (2) It provides an example `LoginModule` implementation for experienced programmers who require the ability to write a `LoginModule` implementing an authentication technology.

`SamplePrincipal.java` is a sample class implementing the `java.security.Principal` interface. If authentication is successful, the `SampleLoginModule` populates a `Subject` with a `SamplePrincipal` representing the user.

Important: If you are an application writer, you do not need to know how to write a `LoginModule` or a `Principal` implementation. You do not need to examine the `SampleLoginModule` or `SamplePrincipal` code. All you have to know is how to write your application and specify configuration information (such as in a login configuration file) such that the application will be able to utilize the `LoginModule` specified by the configuration to authenticate the user. You need to determine which `LoginModule`(s) you want to use and read the `LoginModule`'s documentation to learn about what options you can specify values for (in the configuration) to control the `LoginModule`'s behavior.

Any vendor can provide a LoginModule implementation that you can use. Some implementations are supplied with the JDK from Oracle, as listed in Appendix B: JAAS Login Configuration File.

Information for programmers who want to write a `LoginModule` can be found in Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide.

SampleLoginModule.java

```
package sample.module;

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import sample.principal.SamplePrincipal;

/**
 * <p> This sample LoginModule authenticates users with a password.
 *
 * <p> This LoginModule only recognizes one user:       testUser
 * <p> testUser's password is:  testPassword
 *
 * <p> If testUser successfully authenticates itself,
 * a <code>SamplePrincipal</code> with the testUser's user name
 * is added to the Subject.
 *
 * <p> This LoginModule recognizes the debug option.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 * If set to true in the login Configuration,
 * debug messages will be output to the output stream, System.out.
 *
 */
public class SampleLoginModule implements LoginModule {

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // configurable option
    private boolean debug = false;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // username and password
    private String username;
    private char[] password;

    // testUser's SamplePrincipal
    private SamplePrincipal userPrincipal;

    /**
     * Initialize this <code>LoginModule</code>.
     *
     * @param subject the <code>Subject</code> to be authenticated. <p>
     *
     * @param callbackHandler a <code>CallbackHandler</code> for communicating
     *                  with the end user (prompting for user names and
     *                  passwords, for example). <p>
     *
     * @param sharedState shared <code>LoginModule</code> state. <p>
     *
     * @param options options specified in the login
     *                  <code>Configuration</code> for this particular
     *                  <code>LoginModule</code>.
     */
    public void initialize(Subject subject,
                   CallbackHandler callbackHandler,
                         Map<java.lang.String, ?> sharedState,
                         Map<java.lang.String, ?> options) {

        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;

        // initialize any configured options
        debug = "true".equalsIgnoreCase((String)options.get("debug"));
    }

    /**
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
         * Authenticate the user by prompting for a user name and password.
         *
         * @return true in all cases since this <code>LoginModule</code>
         *          should not be ignored.
         *
         * @exception FailedLoginException if the authentication fails. <p>
         *
         * @exception LoginException if this <code>LoginModule</code>
         *          is unable to perform the authentication.
         */
        public boolean login() throws LoginException {

            // prompt for a user name and password
            if (callbackHandler == null)
                throw new LoginException("Error: no CallbackHandler available " +
                            "to garner authentication information from the user");

            Callback[] callbacks = new Callback[2];
            callbacks[0] = new NameCallback("user name: ");
            callbacks[1] = new PasswordCallback("password: ", false);

            try {
                callbackHandler.handle(callbacks);
                username = ((NameCallback)callbacks[0]).getName();
                char[] tmpPassword =
    ((PasswordCallback)callbacks[1]).getPassword();
                if (tmpPassword == null) {
                    // treat a NULL password as an empty password
                    tmpPassword = new char[0];
                }
                password = new char[tmpPassword.length];
                System.arraycopy(tmpPassword, 0,
                            password, 0, tmpPassword.length);
                ((PasswordCallback)callbacks[1]).clearPassword();

            } catch (java.io.IOException ioe) {
                throw new LoginException(ioe.toString());
            } catch (UnsupportedCallbackException uce) {
                throw new LoginException("Error: " + uce.getCallback().toString()
    +
                    " not available to garner authentication information " +
                    "from the user");
            }

            // print debugging information
            if (debug) {
                System.out.println("\t\t[SampleLoginModule] " +
                                   "user entered user name: " +
                                   username);
                System.out.print("\t\t[SampleLoginModule] " +
                                   "user entered password: ");
                for (int i = 0; i < password.length; i++)
                    System.out.print(password[i]);
                System.out.println();
            }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        // verify the username/password
        boolean usernameCorrect = false;
        boolean passwordCorrect = false;
        if (username.equals("testUser"))
            usernameCorrect = true;
        if (usernameCorrect &&
            password.length == 12 &&
            password[0] == 't' &&
            password[1] == 'e' &&
            password[2] == 's' &&
            password[3] == 't' &&
            password[4] == 'P' &&
            password[5] == 'a' &&
            password[6] == 's' &&
            password[7] == 's' &&
            password[8] == 'w' &&
            password[9] == 'o' &&
            password[10] == 'r' &&
            password[11] == 'd') {

            // authentication succeeded!!!
            passwordCorrect = true;
            if (debug)
                System.out.println("\t\t[SampleLoginModule] " +
                                   "authentication succeeded");
            succeeded = true;
            return true;
        } else {

            // authentication failed -- clean out state
            if (debug)
                System.out.println("\t\t[SampleLoginModule] " +
                                   "authentication failed");
            succeeded = false;
            username = null;
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
            if (!usernameCorrect) {
                throw new FailedLoginException("User Name Incorrect");
            } else {
                throw new FailedLoginException("Password Incorrect");
            }
        }
    }

    /**
     * This method is called if the LoginContext's
     * overall authentication succeeded
     * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
     * succeeded).
     *
     * If this LoginModule's own authentication attempt
     * succeeded (checked by retrieving the private state saved by the
     * <code>login</code> method), then this method associates a
     * <code>SamplePrincipal</code>
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
     * with the <code>Subject</code> located in the
     * <code>LoginModule</code>.  If this LoginModule's own
     * authentication attempted failed, then this method removes
     * any state that was originally saved.
     *
     * @exception LoginException if the commit fails.
     *
     * @return true if this LoginModule's own login and commit
     *          attempts succeeded, or false otherwise.
     */
    public boolean commit() throws LoginException {
        if (succeeded == false) {
            return false;
        } else {
            // add a Principal (authenticated identity)
            // to the Subject

            // assume the user we authenticated is the SamplePrincipal
            userPrincipal = new SamplePrincipal(username);
            if (!subject.getPrincipals().contains(userPrincipal))
                subject.getPrincipals().add(userPrincipal);

            if (debug) {
                System.out.println("\t\t[SampleLoginModule] " +
                                   "added SamplePrincipal to Subject");
            }

            // in any case, clean out state
            username = null;
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;

            commitSucceeded = true;
            return true;
        }
    }

    /**
     * This method is called if the LoginContext's
     * overall authentication failed.
     * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
     * did not succeed).
     *
     * If this LoginModule's own authentication attempt
     * succeeded (checked by retrieving the private state saved by the
     * <code>login</code> and <code>commit</code> methods),
     * then this method cleans up any state that was originally saved.
     *
     * @exception LoginException if the abort fails.
     *
     * @return false if this LoginModule's own login and/or commit attempts
     *          failed, and true otherwise.
     */
    public boolean abort() throws LoginException {
        if (succeeded == false) {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            return false;
        } else if (succeeded == true && commitSucceeded == false) {
            // login succeeded but overall authentication failed
            succeeded = false;
            username = null;
            if (password != null) {
                for (int i = 0; i < password.length; i++)
                    password[i] = ' ';
                password = null;
            }
            userPrincipal = null;
        } else {
            // overall authentication succeeded and commit succeeded,
            // but someone else's commit failed
            logout();
        }
        return true;
    }

    /**
     * Logout the user.
     *
     * This method removes the <code>SamplePrincipal</code>
     * that was added by the <code>commit</code> method.
     *
     * @exception LoginException if the logout fails.
     *
     * @return true in all cases since this <code>LoginModule</code>
     *          should not be ignored.
     */
    public boolean logout() throws LoginException {

        subject.getPrincipals().remove(userPrincipal);
        succeeded = false;
        succeeded = commitSucceeded;
        username = null;
        if (password != null) {
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
        }
        userPrincipal = null;
        return true;
    }
}
```

SamplePrincipal.java

```
package sample.principal;

import java.security.Principal;

/**
 * This class implements the <code>Principal</code> interface
 * and represents a Sample user.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 *
 * Principals such as this <code>SamplePrincipal</code>
 * may be associated with a particular <code>Subject</code>
 * to augment that <code>Subject</code> with an additional
 * identity.  Refer to the <code>Subject</code> class for more information
 * on how to achieve this.  Authorization decisions can then be based upon
 * the Principals associated with a <code>Subject</code>.
 *
 * @see java.security.Principal
 * @see javax.security.auth.Subject
 */
public class SamplePrincipal implements Principal, java.io.Serializable {

    /**
     * @serial
     */
    private String name;

    /**
     * Create a SamplePrincipal with a Sample username.
     *
     * @param name the Sample username for this user.
     *
     * @exception NullPointerException if the <code>name</code>
     *                  is <code>null</code>.
     */
    public SamplePrincipal(String name) {
        if (name == null)
            throw new NullPointerException("illegal null input");

        this.name = name;
    }

    /**
     * Return the Sample username for this <code>SamplePrincipal</code>.
     *
     * @return the Sample username for this <code>SamplePrincipal</code>
     */
    public String getName() {
        return name;
    }

    /**
     * Return a string representation of this <code>SamplePrincipal</code>.
     *
     * @return a string representation of this <code>SamplePrincipal</code>.
     */
    public String toString() {
        return("SamplePrincipal:  " + name);
    }

    /**
     * Compares the specified Object with this <code>SamplePrincipal</code>
     * for equality.  Returns true if the given object is also a
     * <code>SamplePrincipal</code> and the two SamplePrincipals
     * have the same username.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
     *
     * @param o Object to be compared for equality with this
     *          <code>SamplePrincipal</code>.
     *
     * @return true if the specified Object is equal equal to this
     *          <code>SamplePrincipal</code>.
     */
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this == o)
            return true;

        if (!(o instanceof SamplePrincipal))
            return false;
        SamplePrincipal that = (SamplePrincipal)o;

        if (this.getName().equals(that.getName()))
            return true;
        return false;
    }

    /**
     * Return a hash code for this <code>SamplePrincipal</code>.
     *
     * @return a hash code for this <code>SamplePrincipal</code>.
     */
    public int hashCode() {
        return name.hashCode();
    }
}
```

The Login Configuration

JAAS authentication is performed in a pluggable fashion, so applications can remain independent from underlying authentication technologies. A system administrator determines the authentication technologies, or `LoginModules`, to be used for each application and configures them in a login `Configuration`. The source of the configuration information (for example, a file or a database) is up to the current `javax.security.auth.login.Configuration` implementation. The default `Configuration` implementation from Oracle reads configuration information from configuration files, as described in the `ConfigFile` class.

See Appendix B: JAAS Login Configuration File for information as to what a login configuration file is, what it contains, and how to specify which login configuration file should be used.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Login Configuration File for the JAAS Authentication Tutorial

As noted, the login configuration file we use for this tutorial, `sample_jaas.config`, contains just one entry, which is

```
Sample {
  sample.module.SampleLoginModule required debug=true;
};
```

This entry is named "Sample" and that is the name that our tutorial application, `SampleAcn`, uses to refer to this entry. The entry specifies that the `LoginModule` to be used to do the user authentication is the `SampleLoginModule` in the `sample.module` package and that this `SampleLoginModule` is required to "succeed" in order for authentication to be considered successful. The `SampleLoginModule` succeeds only if the name and password supplied by the user are the one it expects ("testUser" and "testPassword", respectively).

The `SampleLoginModule` also defines a "debug" option that can be set to `true` as shown. If this option is set to `true`, `SampleLoginModule` outputs extra information about the progress of authentication. A `LoginModule` can define as many options as it wants. The `LoginModule` documentation should specify the possible option names and values you can set in your configuration file.

Running the JAAS Authentication Tutorial Code

To execute our JAAS authentication tutorial code, all you have to do is

1. Place the following file into a directory:
   - `sample_jaas.config` login configuration file (see The Login Configuration File for the JAAS Authentication Tutorial)
2. Create a subdirectory named `sample` of that top-level directory, and place the following into it (note the `SampleAcn` and `MyCallbackHandler` classes, both in `SampleAcn.java`, are in a package named `sample`):
   - `SampleAcn.java` application source file
3. Create a subdirectory of the `sample` directory and name it `module`. Place the following into it (note the `SampleLoginModule` class is in a package named `sample.module`):
   - `SampleLoginModule.java` source file
4. Create another subdirectory of the `sample` directory and name it `principal`. Place the following into it (note the `SamplePrincipal` class is in a package named `sample.principal`):
   - `SamplePrincipal.java` source file
5. While in the top-level directory, compile `SampleAcn.java`, `SampleLoginModule.java`, and `SamplePrincipal.java`:

   ```
   javac --enable-preview -source 27 sample/SampleAcn.java sample/module/
   SampleLoginModule.java sample/principal/SamplePrincipal.java
   ```

   (Type all that on one line.)
6. Execute the `SampleAcn` application, specifying
   - by `-Djava.security.auth.login.config==sample_jaas.config` that the login configuration file to be used is `sample_jaas.config`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The following is the full command:

```
java --enable-preview -Djava.security.auth.login.config==sample_jaas.config
sample.SampleAcn
```

> **Note**
>
> If you use a single equals sign (=) with the `java.security.auth.login.config` system property (instead of a double equals sign (==)), then the configurations specified by both this system property and the `java.security` file are used.

You will be prompted for your user name and password, and the `SampleLoginModule` specified in the login configuration file will check to ensure these are correct. The `SampleLoginModule` expects `testUser` for the user name and `testPassword` for the password.

You will see some messages output by `SampleLoginModule` as a result of the `debug` option being set to `true` in the login configuration file. Then, if your login is successful, you will see the following message output by `SampleAcn`:

```
Authentication succeeded!
```

If the login is not successful (for example, if you misspell the password), you will see

```
Authentication failed:
```

followed by a reason for the failure. For example, if you mistype the password, you may see a message like the following:

```
Authentication failed:
  Password Incorrect
```

`SampleAcn` gives you three chances to successfully log in.

The sample prints output similar to the following:

```
$ java --enable-preview -Djava.security.auth.login.config==sample_jaas.config
sample.SampleAcn
user name: testUser
password: testPassword
                [SampleLoginModule] user entered user name: testUser
                [SampleLoginModule] user entered password: testPassword
                [SampleLoginModule] authentication succeeded
                [SampleLoginModule] added SamplePrincipal to Subject
Authentication succeeded!
Authenticated user has the following Principals:
        SamplePrincipal:  testUser
User has 0 Public Credential(s)

Current subject: Subject:
        Principal: SamplePrincipal:  testUser
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
Adding principal -1501634219

Authenticated user has the following Principals:

Adding principal -1164915804

Adding principal 304653985

Adding principal -1764420412

Current subject: Subject:
        Principal: SamplePrincipal:  testUser
        Principal: SamplePrincipal:  -1501634219
        Principal: SamplePrincipal:  -1164915804
        Principal: SamplePrincipal:  304653985
        Principal: SamplePrincipal:  -1764420412
```

Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide

JAAS provides subject-based authorization on authenticated identities. This document focuses on the authentication aspect of JAAS, specifically the `LoginModule` interface.

Topics

- Who Should Read This Document
- Related Documentation
- Introduction to LoginModule
- Steps to Implement a LoginModule

Who Should Read This Document

This document is intended for experienced programmers who require the ability to write a `LoginModule` implementing an authentication technology.

Related Documentation

This document assumes you have already read the following:

- Java Authentication and Authorization Service (JAAS) Reference Guide

It also discusses various classes and interfaces in the JAAS API. See the JavaDoc API documentation for the JAAS API specification for more detailed information:

- `javax.security.auth`
- `javax.security.auth.callback`
- `javax.security.auth.kerberos`
- `javax.security.auth.login`
- `javax.security.auth.spi`
- `javax.security.auth.x500`

Copyright © 1993, 2026, Oracle and/or its affiliates.

The following packages contain supported `LoginModule` examples:

- `com.sun.security.auth`
- `com.sun.security.auth.callback`
- `com.sun.security.auth.login`
- `com.sun.security.auth.module`

The JAAS Authentication Tutorial can be run by everyone.

A similar tutorial, JAAS Authentication Tutorial, demonstrates the use of a Kerberos LoginModule. It requires a Kerberos installation. It's part of Introduction to JAAS and Java GSS-API Tutorials that utilize Kerberos as the underlying technology for authentication and secure communication.

Introduction to LoginModule

Authentication technology providers must implement the `LoginModule` interface. `LoginModules` are plugged in under applications to provide a particular type of authentication.

While applications write to the `LoginContext` Application Programming Interface (API), authentication technology providers implement the `LoginModule` interface. A `Configuration` specifies the `LoginModule`(s) to be used with a particular login application. Different `LoginModules` can be plugged in under the application without requiring any modifications to the application itself.

The `LoginContext` is responsible for reading the `Configuration` and instantiating the specified `LoginModules`. Each `LoginModule` is initialized with a `Subject`, a `CallbackHandler`, shared `LoginModule` state, and `LoginModule`-specific options.

The **`Subject`** represents the user or service currently being authenticated and is updated by a `LoginModule` with relevant `Principal`s and credentials if authentication succeeds. `LoginModules` use the **`CallbackHandler`** to communicate with users (to prompt for user names and passwords, for example), as described in the login method description. Note that the `CallbackHandler` may be null. A `LoginModule` that requires a `CallbackHandler` to authenticate the `Subject` may throw a `LoginException` if it was initialized with a `null` `CallbackHandler`. `LoginModules` optionally use the **shared state** to share information or data among themselves.

The **`LoginModule`-specific options** represent the options configured for this `LoginModule` in the login `Configuration`. The options are defined by the `LoginModule` itself and control the behavior within it. For example, a `LoginModule` may define options to support debugging/testing capabilities. Options are defined using a key-value syntax, such as *debug=true*. The `LoginModule` stores the options as a `Map` so that the values may be retrieved using the key. Note that there is no limit to the number of options a `LoginModule` chooses to define.

The calling application sees the authentication process as a single operation invoked via a call to the `LoginContext`'s `login` method. However, the authentication process within each `LoginModule` proceeds in two distinct phases. In the first phase of authentication, the `LoginContext`'s `login` method invokes the `login` method of each `LoginModule` specified in the `Configuration`. The `login` method for a `LoginModule` performs the actual authentication (prompting for and verifying a password for example) and saves its authentication status as private state information. Once finished, the `LoginModule`'s `login` method returns `true` (if it succeeded) or `false` (if it should be ignored), or it throws a `LoginException` to specify a failure. In the failure case, the `LoginModule` must not retry the authentication or introduce delays. The responsibility of such tasks belongs to the application. If the application attempts to retry the authentication, each `LoginModule`'s `login` method will be called again.

Copyright © 1993, 2026, Oracle and/or its affiliates.

In the second phase, if the `LoginContext`'s overall authentication succeeded (calls to the relevant *required*, *requisite*, *sufficient* and *optional* `LoginModules`' `login` methods succeeded), then the `commit` method for each `LoginModule` gets invoked. (For an explanation of the `LoginModule` flags *required*, *requisite*, *sufficient* and *optional*, please consult the `Configuration` documentation and Appendix B: JAAS Login Configuration File in the JAAS Reference Guide.) The `commit` method for a `LoginModule` checks its privately saved state to see if its own authentication succeeded. If the overall `LoginContext` authentication succeeded and the `LoginModule`'s own authentication succeeded, then the `commit` method associates the relevant `Principals` (authenticated identities) and credentials (authentication data such as cryptographic keys) with the `Subject`.

If the `LoginContext`'s overall authentication failed (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL `LoginModules`' `login` methods did not succeed), then the `abort` method for each `LoginModule` gets invoked. In this case, the `LoginModule` removes/destroys any authentication state originally saved.

Logging out a `Subject` involves only one phase. The `LoginContext` invokes the `LoginModule`'s `logout` method. The `logout` method for the `LoginModule` then performs the logout procedures, such as removing `Principals` or credentials from the `Subject`, or logging session information.

Steps to Implement a LoginModule

The following are the steps required to implement and test a `LoginModule`:

- Step 1: Understand the Authentication Technology
- Step 2: Name the LoginModule Implementation
- Step 3: Implement the LoginModule Interface
- Step 4: Choose or Write a Sample Application
- Step 5: Compile the LoginModule and Application
- Step 6: Prepare for Testing
- Step 7: Test Use of the LoginModule
- Step 8: Document Your LoginModule Implementation
- Step 9: Make LoginModule JAR File and Documents Available

Step 1: Understand the Authentication Technology

First, understand the authentication technology to be implemented by your new `LoginModule` provider and determine its requirements.

1. Determine whether or not your `LoginModule` will require some form of user interaction (retrieving a user name and password, for example). If so, you will need to become familiar with the `CallbackHandler` interface and the `javax.security.auth.callback` package.

   In that package you will find several possible `Callback` implementations to use. (Alternatively, you can create your own `Callback` implementations.) The `LoginModule` will invoke the `CallbackHandler` specified by the application itself and passed to the `LoginModule`'s `initialize` method. The `LoginModule` passes the `CallbackHandler` an array of appropriate `Callbacks`. See LoginModule.login Method in Step 3: Implement the LoginModule Interface.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> It is possible for `LoginModule` implementations not to have any end-user interactions. Such `LoginModules` would not need to access the `callback` package.

2. Determine what configuration options you want to make available to the user, who specifies configuration information in whatever form the current Configuration implementation expects (for example, in files). For each option, decide the option name and possible values.

   For example, if a `LoginModule` may be configured to consult a particular authentication server host, decide on the option's key name ("auth_server", for example), as well as the possible server hostnames valid for that option ("server_one.example.com" and "server_two.example.com", for example).

Step 2: Name the LoginModule Implementation

Decide on the proper package and class name for your `LoginModule`.

For example, a `LoginModule` developed by IBM might be called `com.ibm.auth.Module` where `com.ibm.auth` is the package name and `Module` is the name of the `LoginModule` class implementation.

Step 3: Implement the LoginModule Interface

The `LoginModule` interface specifies five abstract methods that you must implement:

- `initialize`
- `login`
- `commit`
- `abort`
- `logout`

In addition to these methods, a `LoginModule` implementation must provide a public constructor with no arguments. This allows for its proper instantiation by a `LoginContext`. Note that if no constructor is provided in your `LoginModule` implementation, a default no-argument constructor is automatically inherited from the `Object` class.

> **Note**
>
> If you don't implement the `LoginModule` interface, then a `LoginException` will be thrown when you try to use your login module.

LoginModule.initialize Method

```
public void initialize(
    Subject subject,
    CallbackHandler handler,
    Map<java.lang.String, ?> sharedState,
    Map<java.lang.String, ?> options);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The `initialize` method is called to initialize the `LoginModule` with the relevant authentication and state information.

This method is called by a `LoginContext` immediately after this `LoginModule` has been instantiated, and prior to any calls to its other public methods. The method implementation should store away the provided arguments for future use.

The `initialize` method may additionally peruse the provided *sharedState* to determine what additional authentication state it was provided by other `LoginModules`, and may also traverse through the provided *options* to determine what configuration options were specified to affect the `LoginModule`'s behavior. It may save option values in variables for future use.

The following is a list of options commonly supported by `LoginModules`. Note that the following is simply a guideline. Modules are free to support a subset (or none) of the following options.

- `tryFirstPass` - If `true`, the first `LoginModule` in the stack saves the password entered, and subsequent `LoginModules` also try to use it. If authentication fails, the `LoginModules` prompt for a new password and retry the authentication.
- `useFirstPass` - If `true`, the first `LoginModule` in the stack saves the password entered, and subsequent `LoginModules` also try to use it. `LoginModules` do not prompt for a new password if authentication fails (authentication simply fails).
- `tryMappedPass` - If `true`, the first `LoginModule` in the stack saves the password entered, and subsequent `LoginModules` attempt to map it into their service-specific password. If authentication fails, the `LoginModules` prompt for a new password and retry the authentication.
- `useMappedPass` - If `true`, the first `LoginModule` in the stack saves the password entered, and subsequent `LoginModules` attempt to map it into their service-specific password. `LoginModules` do not prompt for a new password if authentication fails (authentication simply fails).
- `moduleBanner` - If `true`, then when invoking the `CallbackHandler`, the `LoginModule` provides a `TextOutputCallback` as the first `Callback`, which describes the `LoginModule` performing the authentication.
- `debug` - If `true`, instructs a `LoginModule` to output debugging information.

The `initialize` method may freely ignore state or options it does not understand, although it would be wise to log such an event if it does occur.

Note that the `LoginContext` invoking this `LoginModule` (and the other configured `LoginModules`, as well), all share the same references to the provided `Subject` and `sharedState`. Modifications to the `Subject` and `sharedState` will, therefore, be seen by all.

LoginModule.login Method

```
boolean login() throws LoginException;
```

The `login` method is called to authenticate a `Subject`. This is phase 1 of authentication.

This method implementation should perform the actual authentication. For example, it may cause prompting for a user name and password, and then attempt to verify the password against a password database. Another example implementation may inform the user to insert their finger into a fingerprint reader, and then match the input fingerprint against a fingerprint database.

Copyright © 1993, 2026, Oracle and/or its affiliates.

If your `LoginModule` requires some form of user interaction (retrieving a user name and password, for example), it should not do so directly. That is because there are various ways of communicating with a user, and it is desirable for `LoginModules` to remain independent of the different types of user interaction. Rather, the `LoginModule`'s `login` method should invoke the `handle` method of the `CallbackHandler` interface passed to the `initialize` method to perform the user interaction and set appropriate results, such as the user name and password. The `LoginModule` passes the `CallbackHandler` an array of appropriate `Callbacks`, for example a `NameCallback` for the user name and a `PasswordCallback` for the password, and the `CallbackHandler` performs the requested user interaction and sets appropriate values in the `Callbacks`. For example, to process a `NameCallback`, the `CallbackHandler` may prompt for a name, retrieve the value from the user, and call the `NameCallback`'s `setName` method to store the name.

The authentication process may also involve communication over a network. For example, if this method implementation performs the equivalent of a *kinit* in Kerberos, then it would need to contact the KDC. If a password database entry itself resides in a remote naming service, then that naming service needs to be contacted, perhaps via the Java Naming and Directory Interface (JNDI). Implementations might also interact with an underlying operating system. For example, if a user has already logged into an operating system like Linux, macOS, or Windows, this method might simply import the underlying operating system's identity information.

The `login` method should

1. Determine whether or not this `LoginModule` should be ignored. One example of when it should be ignored is when a user attempts to authenticate under an identity irrelevant to this `LoginModule` (if a user attempts to authenticate as *root* using NIS, for example). If this `LoginModule` should be ignored, `login` should return `false`. Otherwise, it should do the following:
2. Call the `CallbackHandler handle` method if user interaction is required.
3. Perform the authentication.
4. Store the authentication result (success or failure).
5. If authentication succeeded, save any relevant state information that may be needed by the `commit` method.
6. Return `true` if authentication succeeds, or throw a `LoginException` such as `FailedLoginException` if authentication fails.

Note that the `login` method implementation should not associate any new `Principal` or credential information with the saved `Subject` object. This method merely performs the authentication, and then stores away the authentication result and corresponding authentication state. This result and state will later be accessed by the `commit` or `abort` method. Note that the result and state should typically not be saved in the *sharedState* `Map`, as they are not intended to be shared with other `LoginModules`.

An example of where this method might find it useful to store state information in the *sharedState* `Map` is when `LoginModules` are configured to share passwords. In this case, the entered password would be saved as shared state. By sharing passwords, the user only enters the password once, and can still be authenticated to multiple `LoginModules`. The standard conventions for saving and retrieving names and passwords from the *sharedState* `Map` are the following:

- `javax.security.auth.login.name` - Use this as the shared state map key for saving/retrieving a name. The value should be a `String`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `javax.security.auth.login.password` - Use this as the shared state map key for saving/retrieving a password. The value should be a `char` array.

If authentication fails, the `login` method should not retry the authentication. This is the responsibility of the application. Multiple `LoginContext` `login` method calls by an application are preferred over multiple login attempts from within `LoginModule.login()`.

LoginModule.commit Method

```
boolean commit() throws LoginException;
```

The `commit` method is called to commit the authentication process. This is phase 2 of authentication when phase 1 succeeds. It is called if the `LoginContext`'s overall authentication succeeded (that is, if the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL `LoginModules` succeeded.)

This method should access the authentication result and corresponding authentication state saved by the `login` method.

If the authentication result denotes that the `login` method failed, then this `commit` method should remove/destroy any corresponding state that was originally saved.

If the saved result instead denotes that this `LoginModule`'s `login` method succeeded, then the corresponding state information should be accessed to build any relevant `Principal` and credential information. Such `Principals` and credentials should then be added to the `Subject` stored away by the `initialize` method.

After adding `Principals` and credentials, dispensable state fields should be destroyed expeditiously. Likely fields to destroy would be user names and passwords stored during the authentication process.

The `commit` method should save private state indicating whether the commit succeeded or failed.

The following chart depicts what a `LoginModule`'s `commit` method should return. The different boxes represent the different situations that may occur. For example, the top-left corner box depicts what the `commit` method should return if both the previous call to `login` succeeded and the `commit` method itself succeeded.

Table 6-2 LoginModule.commit Method Return Values

Login Status	COMMIT: SUCCESS	COMMIT: FAILURE
LOGIN: SUCCESS	return TRUE	throw EXCEPTION
LOGIN: FAILURE	return FALSE	return FALSE

LoginModule.abort Method

```
boolean abort() throws LoginException;
```

The `abort` method is called to abort the authentication process. This is phase 2 of authentication when phase 1 fails. It is called if the `LoginContext`'s overall authentication failed.

This method first accesses this `LoginModule`'s authentication result and corresponding authentication state saved by the `login` (and possibly `commit`) methods, and then clears out and destroys the information. Sample state to destroy would be user names and passwords.

Copyright © 1993, 2026, Oracle and/or its affiliates.

If this `LoginModule`'s authentication attempt failed, then there shouldn't be any private state to clean up.

The following charts depict what a `LoginModule`'s `abort` method should return. This first chart assumes that the previous call to `login` succeeded. For instance, the `abort` method should return TRUE if both the previous call to `login` and `commit` succeeded, and the `abort` method itself also succeeded.

Table 6-3 LoginModule.abort Method Return Values: Login Succeeded

Login Status	ABORT: SUCCESS	ABORT: FAILURE
COMMIT: SUCCESS	return TRUE	throw EXCEPTION
COMMIT: FAILURE	return TRUE	throw EXCEPTION

The second chart depicts what a `LoginModule`'s abort method should return, assuming that the previous call to login failed. For instance, the `abort` method should return FALSE if the previous call to `login` failed, the previous call to `commit` succeeded, and the `abort` method itself also succeeded.

Table 6-4 LoginModule.abort Method Return Values: Login Failed

Login Status	ABORT: SUCCESS	ABORT: FAILURE
COMMIT: SUCCESS	return FALSE	return FALSE
COMMIT: FAILURE	return FALSE	return FALSE

LoginModule.logout Method

```
boolean logout() throws LoginException;
```

The `logout` method is called to log out a `Subject`.

This method removes `Principals`, and removes/destroys credentials associated with the `Subject` during the `commit` operation. This method should not touch those `Principals` or credentials previously existing in the `Subject`, or those added by other `LoginModules`.

If the `Subject` has been marked *read-only* (the `Subject`'s `isReadOnly` method returns *true*), then this method should only destroy credentials associated with the `Subject` during the `commit` operation (removing the credentials is not possible). If the `Subject` has been marked as *read-only* and the credentials associated with the `Subject` during the `commit` operation are not destroyable (they do not implement the `Destroyable` interface), then this method may throw a `LoginException`.

The `logout` method should return `true` if logout succeeds, or otherwise throw a `LoginException`.

Step 4: Choose or Write a Sample Application

Either choose an existing sample application for your testing, or write a new one.

See Java Authentication and Authorization Service (JAAS) Reference Guide for information about application requirements and a sample application you can use for your testing.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Step 5: Compile the LoginModule and Application

Compile your new `LoginModule` and the application you will use for testing.

Step 6: Prepare for Testing

Step 6a: Place Your `LoginModule` and Application Code in JAR Files

Place your `LoginModule` and application code in separate JAR files. Here is a sample command for creating a JAR file:

```
jar cvf <JAR file name> <list of classes, separated by spaces>
```

This command creates a JAR file with the specified name containing the specified classes.

For more information on the **jar** tool, see `jar`.

Step 6b: Create a Configuration Referencing the LoginModule

Because JAAS supports a pluggable authentication architecture, your new `LoginModule` can be used without requiring modifications to existing applications. Only the login `Configuration` needs to be updated in order to indicate use of a new `LoginModule`.

The default `Configuration` implementation from Oracle reads configuration information from configuration files, as described in `ConfigFile`.

Create a configuration file to be used for testing. For example, to configure the previously-mentioned hypothetical IBM `LoginModule` for an application, the configuration file might look like this:

```
AppName {
    com.ibm.auth.Module REQUIRED debug=true;
};
```

where `AppName` should be whatever name the application uses to refer to this entry in the login configuration file. The application specifies this name as the first argument to the `LoginContext` constructor.

Step 7: Test Use of the LoginModule

Test your application and its use of the `LoginModule`. When you run the application, specify the login configuration file to be used. For example, suppose your application is named `MyApp`, it is located in `MyApp.jar`, and your configuration file is `test.conf`.

You could run the application and specify the configuration file via the following:

```
java -classpath MyApp.jar
 -Djava.security.auth.login.config=test.conf MyApp
```

Type all that on one line. Multiple lines are used here for legibility.

You may want to configure the `LoginModule` with a *debug* option to help ensure that it is working correctly.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Debug your code and continue testing as needed. If you have problems, review the previous steps and ensure they are all completed.

Be sure to vary user input and the `LoginModule` options specified in the configuration file.

Be sure to also include testing using different installation options (e.g., placing the `LoginModule` on the class path or module path).

1. If you find during testing that your `LoginModule` or application needs modifications, make the modifications, recompile (Step 5: Compile the LoginModule and Application).
2. Place the updated code in a JAR file (Step 6a: Place Your `LoginModule` and Application Code in JAR Files).
3. If needed modify the login configuration file (Step 6b: Create a Configuration Referencing the LoginModule).
4. Re-run the application and repeat these steps as needed.

Step 8: Document Your LoginModule Implementation

Write documentation for clients of your `LoginModule`.

Example documentation you may want to include is:

- A README or User Guide describing
  1. The authentication process employed by your `LoginModule` implementation.
  2. Information on how to install the `LoginModule`.
  3. Configuration options accepted by the `LoginModule`. For each option, specify the option name and possible values (or types of values), as well as the behavior the option controls.
- An example `Configuration` file that references your new `LoginModule`.
- API documentation. Putting JavaDoc comments into your source code as you write it will make the JavaDoc API documentation easy to generate.

Step 9: Make LoginModule JAR File and Documents Available

Make your `LoginModule` JAR file and documentation available to clients.

Migrating from the Deprecated for Removal Subject.getSubject and Subject.doAs Methods to the Subject.current and Subject.callAs Methods

> **Note**
>
> In JDK 24, the Security Manager has been permanently disabled. See JEP 486 for more information.

The following methods from the `javax.security.auth.Subject` class have been deprecated for removal because they have dependencies on Security Manager APIs, which have also been deprecated for removal:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `Subject getSubject(AccessControlContext)`
- `T doAs(Subject, PrivilegedAction<T>)`
- `T doAs(Subject, PrivilegedExceptionAction<T>)`
- `T doAsPrivileged(Subject, PrivilegedAction<T>, AccessControlContext)`
- `T doAsPrivileged(Subject, PrivilegedExceptionAction<T>, AccessControlContext)`

Starting in JDK 24, these methods, in addition to `Subject current()` and `T callAs(Subject, Callable<T>)`, behave as follows:

- `getSubject`: The method throws an `UnsupportedOperationException`.
- `current`: The method returns the `Subject` bound to the period of the execution of the current thread.
- `doAs` and `doAsPrivileged`: These methods invoke an action with a `Subject` as the current subject for the bounded period execution of the action.
- `callAs`: Behaves just like `doAs` except the types of its action argument, `Callable<T>`, and thrown exception, `CompletedException`, are different.

Because the Security Manager has been permanently disabled, replace the `Subject.getSubject(AccessControlContext)` and `Subject.doAs` methods with `Subject.current()` and `Subject.callAs(Subject, Callable<T>)`, respectively, in your code. See The callAs and current Methods for Performing an Action as a Particular Subject.

> **Note**
>
> The `Subject.doAsPrivileged` methods don't have replacement APIs.

You can no longer store a `Subject` in an `AccessControlContext`, retrieve it with the `Subject.current()` method, and invoke `AccessController.doPrivileged` with that context.

However, you can access the parent's thread's current `Subject` from the child thread with the `Subject.current()` method with structured concurrency. See Structured Concurrency in *Java Platform, Standard Edition Core Libraries* and The callAs and current Methods for Performing an Action as a Particular Subject for more information.

Check if child threads in your code access the `Subject` of their parent threads through the `AccessControlContext` class. If so, modify your code so that parent threads pass the `Subject` to newly created threads or use structured concurrency.

Copyright © 1993, 2026, Oracle and/or its affiliates.

7

Java Generic Security Services (Java GSS-API)

Java Generic Security Services (Java GSS-API) is used for securely exchanging messages between communicating applications.

Introduction to JAAS and Java GSS-API Tutorials is a series of tutorials demonstrating various aspects of the use of Java Authentication and Authorization Service (JAAS) and Java GSS-API.

Accessing Native GSS-API describes how to enable and configure Java GSS-API to use native GSS-API.

Advanced Security Programming in Java SE Authentication, Secure Communication and Single Sign-On shows you how to use the Java SE GSS APIs to build applications that authenticate their users, communicate securely with other applications and services, and configure your applications in a Kerberos environment to achieve Single Sign-On.

The Kerberos 5 GSS-API Mechanism describes and lists security features regarding Java Generic Security Services (Java GSS) for Kerberos 5.

Troubleshooting JGSS describes how to enable debugging in JGSS. It also describes some problems that may occur when attempting a login and suggestions for solving them.

Introduction to JAAS and Java GSS-API Tutorials

This page links to a series of tutorials demonstrating various aspects of the use of JAAS (Java Authentication and Authorization Service) and Java GSS-API.

JAAS can be used for the *authentication* of users, to reliably and securely determine who is currently executing Java code.

Java GSS-API is used for *securely exchanging messages* between communicating applications. The Java GSS-API contains the Java bindings for the Generic Security Services Application Program Interface (GSS-API) defined in RFC 5653. GSS-API offers application programmers uniform access to security services atop a variety of underlying security mechanisms, including Kerberos.

Note: JSSE is another API that can be used for secure communication. For the differences between the two, see When to Use Java GSS-API Versus JSSE.

The reason both JAAS and Java GSS-API tutorials are presented together is because JAAS authentication is typically performed prior to secure communication using Java GSS-API. Thus JAAS and Java GSS-API are related and often used together. However, it is possible for applications to use JAAS without Java GSS-API, and it is also possible to use Java GSS-API without JAAS.

The following tutorials provide working examples for all of the scenarios described previously.

1. Use of Java GSS-API for Secure Message Exchanges Without JAAS Programming

   Demonstrates the use of the Java GSS-API for secure message exchanges between a client application and a server application.

Copyright © 1993, 2026, Oracle and/or its affiliates.

2. JAAS Authentication

   Explains how an application can authenticate users using JAAS.

3. Use of JAAS Login Utility

   Describes a utility program that authenticates a user using JAAS and executes any application as that user. This utility, as a convenience, essentially performs the operations described in the JAAS Authentication tutorial on your behalf. Therefore it is possible to skip directly to this tutorial if you do not need to know how to perform JAAS authentication directly.

4. Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges

   The most comprehensive tutorial. The Login utility is used to authenticate a service user and to start up a server application as that user. The Login utility is also used to authenticate a client user and to start up a client application as that user. Finally the client and server applications, on behalf of their authenticated client and service users, exchange secure messages using the Java GSS-API.

All applications in all tutorials in this series utilize Kerberos Version 5 as the underlying technology for authentication and secure communication. See Kerberos Requirements. The term "Kerberos" used throughout the tutorials is meant to refer to Kerberos Version 5.

When to Use Java GSS-API Versus JSSE

Java GSS-API and JSSE provide you with the same basic set of security features:

1. Client-server authentication
2. Encryption and integrity protection of transmitted data

However, there are some key differences between the two. This document lists some of them to help you decide which might be more appropriate in your environment:

1. **Kerberos Single Sign-On Support**

   GSS-API contains support for Kerberos as a mandatory security mechanism. This means that if your desktop has Kerberos support, you can write Java GSS-API based applications that never prompt the user for a password.

2. **Communications API**

   JSSE supports a socket-based API. JSSE sockets extend the socket classes found in `java.net` and JSSE socket factories extend the socket factories found in `javax.net`. Thus, if your application is written such that its security needs to be configured via a socket factory, then JSSE might be more appropriate for you. JSSE sockets need to use some reliable transport. Typically, implementations use TCP.

   Java GSS-API, on the other hand, is a token-based API that relies on the application to do the communication. This means that the application can use TCP sockets, UDP datagrams, or any other channel that will allow it to transport Java GSS-API generated tokens. If your application has varying communication protocol needs, then Java GSS-API might be more appropriate for you. Java GSS-API can read and write its tokens using input and output streams. However, you will need to set up the streams yourself.

3. **Credential Delegation**

   Java GSS-API allows the client to delegate its credentials to the server when using Kerberos. If your application will be deployed in a multi-tier environment where intermediaries need to impersonate clients when talking to backend layers, Java GSS-API might be more appropriate for you.

4. **Selective Encryption**

Copyright © 1993, 2026, Oracle and/or its affiliates.

Because Java GSS-API is token-based, you can choose to selectively encrypt certain messages but not all. If your application needs to intersperse plaintext and ciphertext messages, Java GSS-API might be more appropriate for you.

5. **Protocol Requirements**

   JSSE provides implementations of the TLS protocol including TLS version 1.3 and TLS version 1.2. Java GSS-API provides an implementation of the GSS-API framework defined in Generic Security Service API Version 2: Java Bindings Update (RFC 5653), as well as an implementation of the Kerberos Version 5 mechanism defined in The Kerberos Version 5 GSS-API Mechanism (RFC 1964). (On Microsoft Windows platforms, this may be known as SSPI with Kerberos.) Some servers such as HTTPS servers will require you to use TLS, in which case JSSE will be appropriate for you. Other servers such as LDAP servers using SASL might need GSS-API with Kerberos, in which case Java GSS-API will be appropriate for you.

Use of Java GSS-API for Secure Message Exchanges Without JAAS Programming

This tutorial presents two sample applications demonstrating the use of the Java GSS-API for secure exchanges of messages between communicating applications, in this case a client application and a server application.

Java GSS-API uses what is called a "security mechanism" to provide these services. The GSS-API implementation contains support for the Kerberos V5 mechanism in addition to any other vendor-specific choices. The Kerberos V5 mechanism is used for this tutorial.

In order to perform authentication between the client and server and to establish cryptographic keys for secure communication, a GSS-API mechanism needs access to certain credentials for the local entity on each side of the connection. In our case, the credential used on the client side consists of a Kerberos ticket, and on the server side, it consists of a long-term Kerberos secret key. Kerberos tickets can optionally include the host address and IPv4 and IPv6 host addresses are both supported. Java GSS-API requires that the mechanism obtain these credentials from the `Subject` bound to the period of the execution of the current thread.

To populate a `Subject` with such credentials, client and server applications typically will first perform JAAS authentication using a Kerberos `LoginModule`. The JAAS Authentication tutorial demonstrates how to do this. A utility has also been written as a convenience to automatically perform those operations on your behalf. The Use of JAAS Login Utility tutorial demonstrates how to use the Login utility.

For this tutorial, we will not have the client and server perform JAAS authentication, nor will we have them use the Login utility. Instead, we will rely on setting the system property `javax.security.auth.useSubjectCredsOnly` to `false`, which allows us to relax the restriction of requiring a GSS mechanism to obtain necessary credentials from an existing Subject, set up by JAAS. See The useSubjectCredsOnly System Property.

> **Note**
>
> This is a simplified introductory tutorial.

There is another tutorial, Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges, that is just like the tutorial you are reading except that it utilizes the Login utility and a more complex login configuration file. A login configuration file (see Appendix B: JAAS

Copyright © 1993, 2026, Oracle and/or its affiliates.

Login Configuration File), required whenever JAAS authentication is done, specifies the desired authentication module.

As with all tutorials in this series, the underlying technology used to support authentication and secure communication for the applications in this tutorial is Kerberos V5. See Kerberos Requirements.

- Overview of the Client and Server Applications
- The SampleClient and SampleServer Code
- Kerberos User and Service Principal Names
- The Login Configuration File
- The useSubjectCredsOnly System Property
- Running the SampleClient and SampleServer Programs

If you want to first see the tutorial code in action, you can skip directly to Running the SampleClient and SampleServer Programs and then go back to the other sections to learn more.

Overview of the Client and Server Applications

The applications for this tutorial are named `SampleClient` and `SampleServer`.

Here is a summary of execution of the `SampleClient` and `SampleServer` applications:

1. Run the `SampleServer` application. `SampleServer`
   a. Reads its argument, the port number that it should listen on for client connections.
   b. Creates a `ServerSocket` for listening for client connections on that port.
   c. Listens for a connection.
2. Run the `SampleClient` application (possibly on a different machine). `SampleClient`
   a. Reads its arguments: (1) The name of the Kerberos principal that represents `SampleServer` (see Kerberos User and Service Principal Names), (2) the name of the host (machine) on which `SampleServer` is running, and (3) the port number on which `SampleServer` listens for client connections.
   b. Attempts a socket connection with the `SampleServer`, using the host and port it was passed as arguments.
3. The socket connection is accepted by `SampleServer` and both applications initialize a `DataInputStream` and a `DataOutputStream` from the socket input and output streams, to be used for future data exchanges.
4. `SampleClient` and `SampleServer` each instantiate a `GSSContext` and follow a protocol for establishing a shared context that will enable subsequent secure data exchanges.
5. `SampleClient` and `SampleServer` can now securely exchange messages.
6. When `SampleClient` and `SampleServer` are done exchanging messages, they perform clean-up operations.

The actual code and further details are presented in the following sections.

The SampleClient and SampleServer Code

The entire code for both the `SampleClient.java` and `SampleServer.java` programs resides in their `main` methods and can be broken down into the following subparts:

Copyright © 1993, 2026, Oracle and/or its affiliates.

1. Obtaining the Command-Line Arguments
2. Establishing a Socket Connection for Message Exchanges
3. Establishing a Security Context
4. Exchanging Messages Securely
5. Clean Up

> **Note**
>
> The Java GSS-API classes utilized by these programs (`GSSManager`, `GSSContext`, `GSSName`, `GSSCredential`, `MessageProp`, and `Oid`) are found in the `org.ietf.jgss` package.

Obtaining the Command-Line Arguments

The first thing both our client and server `main` methods do is read the command-line arguments.

Arguments Read by SampleClient

`SampleClient` expects three arguments:

1. A service principal name – The name of the Kerberos principal that represents `SampleServer` (see Kerberos User and Service Principal Names).
2. A host name – The machine on which `SampleServer` is running.
3. A port number – The port number of the port on which `SampleServer` listens for connections.

Here is the code for reading the command-line arguments:

```
if (args.length < 3) {
    System.out.println("Usage: java <options> Login SampleClient "
       + " <servicePrincipal> <hostName> <port>");
    System.exit(-1);
}

String server = args[0];
String hostName = args[1];
int port = Integer.parseInt(args[2]);
```

Argument Read by SampleServer

`SampleServer` expects just one argument:

- A local port number – The port number used by `SampleServer` for listening for connections with clients. This number should be the same as the port number specified when running the `SampleClient` program.

Here is the code for reading the command-line argument:

```
if (args.length != 1) {
    System.out.println(
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            "Usage: java <options> Login SampleServer <localPort>");
        System.exit(-1);
    }

    int localPort = Integer.parseInt(args[0]);
```

Establishing a Socket Connection for Message Exchanges

Java GSS-API provides methods for creating and interpreting tokens (opaque byte data). The tokens contain messages to be securely exchanged between two peers, but the method of actual token transfer is up to the peers. For our `SampleClient` and `SampleServer` applications, we establish a socket connection between the client and server and exchange data using the socket input and output streams.

SampleClient Code for Socket Connection

`SampleClient` was passed as arguments the name of the host machine `SampleServer` is running on, as well as the port number on which `SampleServer` will be listening for connections, so `SampleClient` has all it needs to establish a socket connection with `SampleServer`. It uses the following code to set up the connection and initialize a `DataInputStream` and a `DataOutputStream` for future data exchanges:

```
Socket socket = new Socket(hostName, port);

DataInputStream inStream =
  new DataInputStream(socket.getInputStream());
DataOutputStream outStream =
  new DataOutputStream(socket.getOutputStream());

System.out.println("Connected to server "
   + socket.getInetAddress());
```

SampleServer Code for Socket Connection

The `SampleServer` application was passed as an argument the port number to be used for listening for connections from clients. It creates a `ServerSocket` for listening on that port:

```
ServerSocket ss = new ServerSocket(localPort);
```

The `ServerSocket` can then wait for and accept a connection from a client, and then initialize a `DataInputStream` and a `DataOutputStream` for future data exchanges with the client :

```
Socket socket = ss.accept();

DataInputStream inStream =
    new DataInputStream(socket.getInputStream());
DataOutputStream outStream =
    new DataOutputStream(socket.getOutputStream());

System.out.println("Got connection from client "
    + socket.getInetAddress());
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The `accept` method waits until a client (in our case, `SampleClient`) requests a connection on the host and port of the `SampleServer`, which `SampleClient` does via

```
Socket socket = new Socket(hostName, port);
```

When the connection is requested and established, the `accept` method returns a new `Socket` object bound to a new port. The server can communicate with the client over this new socket and continue to listen for other client connection requests on the `ServerSocket` bound to the original port. Thus, a server program typically has a loop which can handle multiple connection requests.

The basic loop structure for our `SampleServer` is the following:

```
while (true) {

    Socket socket = ss.accept();

    <Establish input and output streams for the connection>;
    <Establish a context with the client>;
    <Exchange messages with the client>;
    <Clean up>;
}
```

Client connections are queued at the original port, so with this program structure used by `SampleServer`, the interaction with the first client making a connection has to complete before the next connection can be accepted. The server could actually service multiple clients simultaneously through the use of threads – one thread per client connection, as in

```
while (true) {
    <accept a connection>;
    <create a thread to handle the client>;
}
```

Establishing a Security Context

Before two applications can use Java GSS-API to securely exchange messages between them, they must establish a joint security context using their credentials. (Note: In the case of `SampleClient`, the credentials were established when the Login utility authenticated the user on whose behalf the `SampleClient` was run, and similarly for `SampleServer`.) The security context encapsulates shared state information that might include, for example, cryptographic keys. One use of such keys might be to encrypt messages to be exchanged, if encryption is requested.

As part of the security context establishment, the context initiator (in our case, `SampleClient`) is authenticated to the acceptor (`SampleServer`), and may require that the acceptor also be authenticated back to the initiator, in which case we say that "mutual authentication" took place.

Both applications create and use a `GSSContext` object to establish and maintain the shared information that makes up the security context.

The instantiation of the context object is done differently by the context initiator and the context acceptor. After the initiator instantiates a `GSSContext`, it may choose to set various context options that will determine the characteristics of the desired security context, for example, specifying whether or not mutual authentication should take place. After all the desired

Copyright © 1993, 2026, Oracle and/or its affiliates.

characteristics have been set, the initiator calls the `initSecContext` method, which produces a token required by the acceptor's `acceptSecContext` method.

While Java GSS-API methods exist for preparing tokens to be exchanged between applications, *it is the responsibility of the applications to actually transfer the tokens between them.* So after the initiator has received a token from its call to `initSecContext`, it sends that token to the acceptor. The acceptor calls `acceptSecContext`, passing it the token. The `acceptSecContext` method may in turn return a token. If it does, the acceptor should send that token to the initiator, which should then call `initSecContext` again and pass it this token. Each time `initSecContext` or `acceptSecContext` returns a token, the application that called the method should send the token to its peer and that peer should pass the token to its appropriate method (`acceptSecContext` or `initSecContext`). This continues until the context is fully established (which is the case when the context's `isEstablished` method returns `true`).

The context establishment code for our sample applications is described in the following:

- Context Establishment by SampleClient
- Context Establishment by SampleServer

Context Establishment by SampleClient

In our client/server scenario, SampleClient is the context initiator. Here are the basic steps it takes to establish a security context:

1. SampleClient GSSContext Instantiation
2. SampleClient Setting of Desired Options
3. SampleClient Context Establishment Loop: Loops while the context is not yet established, each time calling `initSecContext`, sending any returned token to `SampleServer`, and receiving a token (if any) from `SampleServer`.

SampleClient GSSContext Instantiation

A `GSSContext` is created by instantiating a `GSSContext` and then calling one of its `createContext` methods. The `GSSManager` class serves as a factory for other important GSS API classes. It can create instances of classes implementing the `GSSContext`, `GSSCredential`, and `GSSName` interfaces.

`SampleClient` obtains an instance of the default `GSSManager` subclass by calling the `GSSManager` static method `getInstance`:

```
GSSManager manager = GSSManager.getInstance();
```

The default `GSSManager` subclass is one whose `create*` methods (`createContext`, etc.) return classes whose implementations support Kerberos as the underlying technology.

The `GSSManager` factory method for creating a context on the initiator's side has the following signature:

```
GSSContext createContext(GSSName peer, Oid mech,
            GSSCredential myCred, int lifetime);
```

The following sections describe the arguments, followed by the complete call to `createContext`.

The GSSName peer Argument

The peer in our client/server paradigm is the server. For the `peer` argument, we need a `GSSName` for the service principal representing the server. (See Kerberos User and Service

Copyright © 1993, 2026, Oracle and/or its affiliates.

Principal Names.) A `String` for the service principal name is passed as the first argument to `SampleClient`, which places the argument into its local `String` variable named `server`. The `GSSManager` `manager` is used to instantiate a `GSSName` by calling one of its `createName` methods. `SampleClient` calls the `createName` method with the following signature:

```
GSSName createName(String nameStr, Oid nameType);
```

`SampleClient` passes the `server` `String` for the `nameStr` argument.

The second argument is an `Oid`. An `Oid` represents a Universal Object Identifier. Oids are hierarchically globally-interpretable identifiers used within the GSS-API framework to identify mechanisms and name types. The structure and encoding of Oids is defined in the ISOIEC-8824 and ISOIEC-8825 standards. The `Oid` passed to the `createName` method is specifically a name type `Oid` (not a mechanism Oid).

In GSS-API, string names are often mapped from a mechanism-independent format into a mechanism-specific format. Usually, an Oid specifies what name format the string is in so that the mechanism knows how to do this mapping. Passing in a `null` `Oid` indicates that the name is already in a native format that the mechanism uses. This is the case for the `server` String; it is in the appropriate format for a Kerberos Version 5 name. Thus, `SampleClient` passes a `null` for the `Oid`. Here is the call:

```
GSSName serverName = manager.createName(server, null);
```

The Oid mech Argument

The second argument to the `GSSManager` `createContext` method is an Oid representing the mechanism to be used for the authentication between the client and the server during context establishment and for subsequent secure communication between them.

Our tutorial will use Kerberos V5 as the security mechanism. The Oid for the Kerberos V5 mechanism is defined in RFC 1964 as "1.2.840.113554.1.2.2" so we create such an `Oid`:

```
Oid krb5Oid = new Oid("1.2.840.113554.1.2.2");
```

`SampleClient` passes `krb5Oid` as the second argument to `createContext`.

The GSSCredential myCred Argument

The third argument to the `GSSManager` `createContext` method is a `GSSCredential` representing the caller's credentials. If you pass `null` for this argument, as `SampleClient` does, the default credentials are used.

The int lifetime Argument

The final argument to the `GSSManager` `createContext` method is an `int` specifying the desired lifetime, in seconds, for the context that is created. `SampleClient` passes `GSSContext.DEFAULT_LIFETIME` to request a default lifetime.

The Complete createContext Call

Now that we have all the required arguments, here is the call `SampleClient` makes to create a `GSSContext`:

```
GSSContext context =
    manager.createContext(serverName,
                          krb5Oid,
                          null,
                          GSSContext.DEFAULT_LIFETIME);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

SampleClient Setting of Desired Options

After instantiating a context, and prior to actually establishing the context with the context acceptor, the context initiator may choose to set various options that determine the desired security context characteristics. Each such option is set by calling a `request` method on the instantiated context. Most `request` methods take a `boolean` argument for indicating whether or not the feature is requested. It is not always possible for a request to be satisfied, so whether or not it was can be determined after context establishment by calling one of the `get` methods.

`SampleClient` requests the following:

1. **Mutual authentication**. The context initiator is always authenticated to the acceptor. If the initiator requests mutual authentication, then the acceptor is also authenticated to the initiator.
2. **Confidentiality**. Requesting confidentiality means that you request the *enabling* of encryption for the context method named `wrap`. Encryption is actually used only if the `MessageProp` object passed to the `wrap` method requests privacy.
3. **Integrity**. This requests integrity for the `wrap` and `getMIC` methods. When integrity is requested, a cryptographic tag known as a Message Integrity Code (MIC) will be generated when calling those methods. When `getMIC` is called, the generated MIC appears in the returned token. When `wrap` is called, the MIC is packaged together with the message (the original message or the result of encrypting the message, depending on whether confidentiality was applied) all as part of one token. You can subsequently verify the MIC against the message to ensure that the message has not been modified in transit.

The `SampleClient` code for making these requests on the `GSSException` `context` is the following:

```
context.requestMutualAuth(true);  // Mutual authentication
context.requestConf(true);  // Will use encryption later
context.requestInteg(true); // Will use integrity later
```

After the context is established, the client must explicitly check the context states by calling the accesor methods, like `getMutualAuthState`, `getConfState`, or `getIntegState`, and destroy the security context if any of them do not match the desired state.

> **Note**
>
> When using the default `GSSManager` implementation and the Kerberos mechanism, these requests will always be granted.

SampleClient Context Establishment Loop

After `SampleClient` has instantiated a `GSSContext` and specified the desired context options, it can actually establish the security context with `SampleServer`. To do so, `SampleClient` has a loop. Each loop iteration

1. Calls the context's `initSecContext` method. If this is the first call, the method is passed a `null` token. Otherwise, it is passed the token most recently sent to `SampleClient` by `SampleServer` (a token generated by a `SampleServer` call to `acceptSecContext`).
2. Sends the token returned by `initSecContext` (if any) to `SampleServer`. The first call to `initSecContext` always produces a token. The last call might not return a token.
3. Checks to see if the context is established. If not, `SampleClient` receives another token from `SampleServer` and then starts the next loop iteration.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The tokens returned by `initSecContext` or received from `SampleServer` are placed in a byte array. Tokens should be treated by `SampleClient` and `SampleServer` as opaque data to be passed between them and interpreted by Java GSS-API methods.

The `initSecContext` arguments are a byte array containing a token, the starting offset into that array of where the token begins, and the token length. For the first call, `SampleClient` passes a null token, since no token has yet been received from `SampleServer`.

To exchange tokens with `SampleServer`, `SampleClient` uses the `DataInputStream` `inStream` and `DataOutputStream` `outStream` it previously set up using the input and output streams for the socket connection made with `SampleServer`. Note that whenever a token is written, the number of bytes in the token is written first, followed by the token itself. The reasons are discussed in the introduction to the The SampleClient and SampleServer Message Exchanges section.

Here is the `SampleClient` context establishment loop, followed by code displaying information about who the client and server are and whether or not mutual authentication actually took place:

```
byte[] token = new byte[0];

while (!context.isEstablished()) {

    // token is ignored on the first call
    token = context.initSecContext(token, 0, token.length);

    // Send a token to the server if one was generated by
    // initSecContext
    if (token != null) {
        System.out.println("Will send token of size "
                   + token.length + " from initSecContext.");
        outStream.writeInt(token.length);
        outStream.write(token);
        outStream.flush();
    }

    // If the client is done with context establishment
    // then there will be no more tokens to read in this loop
    if (!context.isEstablished()) {
        token = new byte[inStream.readInt()];
        System.out.println("Will read input token of size "
                   + token.length
                   + " for processing by initSecContext");
        inStream.readFully(token);
    }
}

System.out.println("Context Established! ");
System.out.println("Client is " + context.getSrcName());
System.out.println("Server is " + context.getTargName());
if (context.getMutualAuthState())
    System.out.println("Mutual authentication took place!");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Context Establishment by SampleServer

In our client/server scenario, `SampleServer` is the context acceptor. Here are the basic steps it takes to establish a security context:

1. SampleServer GSSContext Instantiation
2. SampleClient Context Establishment Loop: Loops while the context is not yet established, each time receiving a token from `SampleClient`, calling `acceptSecContext` and passing it the token, and sending any returned token to `SampleClient`.

SampleServer GSSContext Instantiation

As described in SampleClient GSSContext Instantiation, a `GSSContext` is created by instantiating a `GSSManager` and then calling one of its `createContext` methods.

Like `SampleClient`, `SampleServer` obtains an instance of the default `GSSManager` subclass by calling the `GSSManager` static method `getInstance`:

```
GSSManager manager = GSSManager.getInstance();
```

The `GSSManager` factory method for creating a context on the acceptor's side has the following signature:

```
GSSContext createContext(GSSCredential myCred);
```

If you pass `null` for the `GSSCredential` argument, as `SampleServer` does, the default credentials are used. The context is instantiated via the following:

```
GSSContext context = manager.createContext((GSSCredential)null);
```

SampleServer Context Establishment Loop

After SampleServer has instantiated a GSSContext, it can establish the security context with SampleClient. To do so, SampleServer has a loop that continues until the context is established. Each loop iteration does the following:

1. Receives a token from SampleClient. This token is the result of a SampleClient `initSecContext` call.
2. Calls the context's `acceptSecContext` method, passing it the token just received.
3. If `acceptSecContext` returns a token, then SampleServer sends this token to SampleClient and then starts the next loop iteration if the context is not yet established.

The tokens returned by `acceptSecContext` or received from SampleClient are placed in a byte array.

The `acceptSecContext` arguments are a byte array containing a token, the starting offset into that array of where the token begins, and the token length.

To exchange tokens with SampleClient, SampleServer uses the DataInputStream `inStream` and DataOutputStream `outStream` it previously set up using the input and output streams for the socket connection made with SampleClient.

Here is the SampleServer context establishment loop:

```
byte[] token = null;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
while (!context.isEstablished()) {

    token = new byte[inStream.readInt()];
    System.out.println("Will read input token of size "
       + token.length
       + " for processing by acceptSecContext");
    inStream.readFully(token);

    token = context.acceptSecContext(token, 0, token.length);

    // Send a token to the peer if one was generated by
    // acceptSecContext
    if (token != null) {
        System.out.println("Will send token of size "
           + token.length
           + " from acceptSecContext.");
        outStream.writeInt(token.length);
        outStream.write(token);
        outStream.flush();
    }
}

System.out.print("Context Established! ");
System.out.println("Client is " + context.getSrcName());
System.out.println("Server is " + context.getTargName());
if (context.getMutualAuthState())
    System.out.println("Mutual authentication took place!");
```

Exchanging Messages Securely

Once a security context has been established between `SampleClient` and `SampleServer`, they can use the context to securely exchange messages.

- GSSContext Methods for Message Exchange
- The SampleClient and SampleServer Message Exchanges

GSSContext Methods for Message Exchange

Two types of methods exist for preparing messages for secure exchange: `wrap` and `getMIC`. There are actually two `wrap` methods (and two `getMIC` methods), where the differences between the two are the indication of where the input message is (a byte array or an input stream) and where the output should go (to a byte array return value or to an output stream).

The following sections describe these methods for preparing messages for exchange and the corresponding methods for interpretation by the peer of the resulting tokens.

wrap

The `wrap` method is the primary method for message exchanges.

The signature for the `wrap` method called by `SampleClient` is the following:

```
byte[] wrap (byte[] inBuf, int offset, interface len,
                MessageProp msgProp)
```

You pass `wrap` a message (in `inBuf`), the offset into `inBuf` where the message begins (`offset`), and the length of the message (`len`). You also pass a `MessageProp`, which is used

Copyright © 1993, 2026, Oracle and/or its affiliates.

to indicate the desired QOP (Quality-of-Protection) and to specify whether or not privacy (encryption) is desired. A QOP value selects the cryptographic integrity and encryption (if requested) algorithm(s) to be used. The algorithms corresponding to various QOP values are specified by the provider of the underlying mechanism. For example, the values for Kerberos V5 are defined in RFC 1964 in section 4.2. It is common to specify 0 as the QOP value to request the default QOP.

The `wrap` method returns a token containing the message and a cryptographic Message Integrity Code (MIC) over it. The message placed in the token will be encrypted if the `MessageProp` indicates privacy is desired. You do not need to know the format of the returned token; it should be treated as opaque data. You send the returned token to your peer application, which calls the `unwrap` method to "unwrap" the token to get the original message and to verify its integrity.

getMIC

If you simply want to get a token containing a cryptographic Message Integrity Code (MIC) for a supplied message, you call `getMIC`. A sample reason you might want to do this is to confirm with your peer that you both have the same data, by just transporting a MIC for that data without incurring the cost of transporting the data itself to each other.

The signature for the `getMIC` method called by `SampleServer` is the following:

```
byte[] getMIC (byte[] inMsg, int offset, int len,
            MessageProp msgProp)
```

You pass `getMIC` a message (in `inMsg`), the offset into `inMsg` where the message begins (`offset`), and the length of the message (`len`). You also pass a `MessageProp`, which is used to indicate the desired QOP (Quality-of-Protection). It is common to specify 0 as the QOP value to request the default QOP.

If you have a token created by `getMIC` and the message used to calculate the MIC (or a message purported to be the message on which the MIC was calculated), you can call the `verifyMIC` method to verify the MIC for the message. If the verification is successful (that is, if a `GSSException` is not thrown), it proves that the message is exactly the same as it was when the MIC was calculated. A peer receiving a message from an application typically expects a MIC as well, so that they can verify the MIC and be assured the message has not been modified or corrupted in transit. Note: If you know ahead of time that you will want the MIC as well as the message then it is more convenient to use the `wrap` and `unwrap` methods. But there could be situations where the message and the MIC are received separately.

The signature for the `verifyMIC` corresponding to the `getMIC` shown previously is the following:

```
void verifyMIC (byte[] inToken, int tokOffset, int tokLen,
        byte[] inMsg, int msgOffset, int msgLen,
        MessageProp msgProp);
```

This verifies the MIC contained in the `inToken` (of length `tokLen`, starting at offset `tokOffset`) over the message contained in `inMsg` (of length `msgLen`, starting at offset `msgOffset`). The `MessageProp` is used by the underlying mechanism to return information to the caller, such as the QOP indicating the strength of protection that was applied to the message.

The SampleClient and SampleServer Message Exchanges

The message exchanges between `SampleClient` and `SampleServer` are the "standard" steps used for verifying a GSS-API client and server. A group at MIT has written a GSS-API client and a GSS-API server that have become fairly popular test programs for checking

Copyright © 1993, 2026, Oracle and/or its affiliates.

interoperability between different implementations of the GSS-API library. (These GSS-API sample applications can be downloaded as a part of the Kerberos distribution available from MIT at http://web.mit.edu/kerberos.) This client and server from MIT follow the protocol that once the context is established, the client sends a message across and it expects back the MIC on that message. If you implement a GSS-API library, it is common practice to test it by running either the client or server using your library implementation against a corresponding peer server or client that uses another GSS-API library implementation. If both library implementations conform to the standards, then the two peers will be able to communicate successfully.

One implication of testing your client or server against ones written in C (like the MIT ones) is the way tokens must be exchanged. C implementations of GSS-API do not include stream-based methods. In the absence of stream-based methods on your peer, when you write a token you must first write the number of bytes and then write the token. Similarly, when you are reading a token, you first read the number of bytes and then read the token. This is what `SampleClient` and `SampleServer` do.

Here is the summary of the `SampleClient` and `SampleServer` message exchanges:

1. `SampleClient` calls `wrap` to encrypt and calculate a MIC for a message.
2. `SampleClient` sends the token returned from `wrap` to `SampleServer`.
3. `SampleServer` calls `unwrap` to obtain the original message and verify its integrity.
4. `SampleServer` calls `getMIC` to calculate a MIC on the decrypted message.
5. `SampleServer` sends the token returned by `getMIC` (which contains the MIC) to `SampleClient`.
6. `SampleClient` calls `verifyMIC` to verify that the MIC sent by `SampleServer` is a valid MIC for the original message.

SampleClient Code to Encrypt the Message and Send It

The `SampleClient` code for encrypting a message, calculating a MIC for it, and sending the result to `SampleServer` is the following:

```
byte[] messageBytes = "Hello There!\0".getBytes();

/*
 * The first MessageProp argument is 0 to request
 * the default Quality-of-Protection.
 * The second argument is true to request
 * privacy (encryption of the message).
 */
MessageProp prop =  new MessageProp(0, true);

/*
 * Encrypt the data and send it across. Integrity protection
 * is always applied, irrespective of encryption.
 */
token = context.wrap(messageBytes, 0, messageBytes.length,
    prop);
System.out.println("Will send wrap token of size "
    + token.length);
outStream.writeInt(token.length);
outStream.write(token);
outStream.flush();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

SampleServer Code to Unwrap Token, Calculate MIC, and Send It

The following `SampleServer` code reads the wrapped token sent by `SampleClient` and "unwraps" it to obtain the original message and have its integrity verified. The unwrapping in this case includes decryption since the message was encrypted.

> **Note**
>
> Here, the integrity check is expected to succeed. But note that in general if an integrity check fails, it signifies that the message was changed in transit. If the `unwrap` method encounters an integrity check failure, it throws a `GSSException` with major error code `GSSException.BAD_MIC`.

```
/*
 * Create a MessageProp which unwrap will use to return
 * information such as the Quality-of-Protection that was
 * applied to the wrapped token, whether or not it was
 * encrypted, etc. Since the initial MessageProp values
 * are ignored, it doesn't matter what they are set to.
 */
MessageProp prop = new MessageProp(0, false);

/*
 * Read the token. This uses the same token byte array
 * as that used during context establishment.
 */
token = new byte[inStream.readInt()];
System.out.println("Will read token of size "
    + token.length);
inStream.readFully(token);

byte[] bytes = context.unwrap(token, 0, token.length, prop);
String str = new String(bytes);
System.out.println("Received data \""
    + str + "\" of length " + str.length());
System.out.println("Encryption applied: "
    + prop.getPrivacy());
```

Next, `SampleServer` generates a MIC for the decrypted message and sends it to `SampleClient`. This is not really necessary but simply illustrates generating a MIC on the decrypted message, which should be exactly the same as the original message `SampleClient` wrapped and sent to `SampleServer`. When `SampleServer` generates this and sends it to `SampleClient`, and `SampleClient` verifies it, this proves to `SampleClient` that the decrypted message `SampleServer` has is in fact exactly the same as the original message from `SampleClient`.

```
/*
 * First reset the QOP of the MessageProp to 0
 * to ensure the default Quality-of-Protection
 * is applied.
 */
prop.setQOP(0);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
token = context.getMIC(bytes, 0, bytes.length, prop);

System.out.println("Will send MIC token of size "
                   + token.length);
outStream.writeInt(token.length);
outStream.write(token);
outStream.flush();
```

SampleClient Code to Verify the MIC

The following `SampleClient` code reads the MIC calculated by `SampleServer` on the decrypted message and then verifies that the MIC is a MIC for the original message, proving that the decrypted message `SampleServer` has is the same as the original message:

```
token = new byte[inStream.readInt()];
System.out.println("Will read token of size " + token.length);
inStream.readFully(token);

/*
 * Recall messageBytes is the byte array containing
 * the original message and prop is the MessageProp
 * already instantiated by SampleClient.
 */
context.verifyMIC(token, 0, token.length,
          messageBytes, 0, messageBytes.length,
          prop);

System.out.println("Verified received MIC for message.");
```

Clean Up

When `SampleClient` and `SampleServer` have finished exchanging messages, they need to perform cleanup operations. Both contain the following code to

- close the socket connection and
- release system resources and cryptographic information stored in the context object and then invalidate the context.

```
socket.close();
context.dispose();
```

Kerberos User and Service Principal Names

Since the underlying authentication and secure communication technology used by this tutorial is Kerberos V5, we use Kerberos-style principal names wherever a user or service is called for (see Principals).

For example, when you run `SampleClient` you are asked to provide your **user name**. Your Kerberos-style user name is simply the user name you were assigned for Kerberos authentication. It consists of a base user name (like `mjones`) followed by an "@" and your realm (like `mjones@KRBNT-OPERATIONS.EXAMPLE.COM`).

A server program like `SampleServer` is typically considered to offer a "service" and to be run on behalf of a particular "**service principal**." A service principal name for `SampleServer` is needed in several places:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- When you run `SampleServer`, and `SampleClient` attempts a connection to it, the underlying Kerberos mechanism will attempt to authenticate to the Kerberos KDC. It prompts you to log in. You should log in as the appropriate service principal.
- When you run `SampleClient`, one of the arguments is the service principal name. This is needed so `SampleClient` can initiate establishment of a security context with the appropriate service.

Throughout this document, and in the accompanying login configuration file, `service_principal@your_realm`, is used as a placeholder to be replaced by the actual name to be used in your environment. *Any* Kerberos principal can actually be used for the service principal name. So **for the purposes of trying out this tutorial, you could use your user name as both the client user name and the service principal name.**

In a production environment, system administrators typically like servers to be run as specific principals only and may assign a particular name to be used. Often the Kerberos-style service principal name assigned is of the form

```
service_name/machine_name@realm;
```

For example, an nfs service run on a machine named `raven` in the realm named `KRBNT-OPERATIONS.EXAMPLE.COM` could have the service principal name

```
nfs/raven@KRBNT-OPERATIONS.EXAMPLE.COM
```

Such multi-component names are not required, however. Single-component names, just like those of user principals, can be used. For example, an installation might use the same ftp service principal `ftp@realm` for all ftp servers in that realm, while another installation might have different ftp principals for different ftp servers, such as `ftp/host1@realm` and `ftp/host2@realm` on machines `host1` and `host2`, respectively.

When the Realm Is Required in Principal Names

If the realm of a user or service principal name is the default realm (see Kerberos Requirements), you can leave off the realm when you are logging into Kerberos (that is, when you are prompted for your user name). Thus, for example, if your user name is `mjones@KRBNT-OPERATIONS.EXAMPLE.COM`, and you run `SampleClient`, when it requests your user name you could just specify `mjones`, leaving off the realm. The name is interpreted in the context of being a Kerberos principal name and the default realm is appended, as needed.

You can also leave off the realm if a principal name will be converted to a `GSSName` by a `GSSManager` `createName` method. For example, when you run `SampleClient`, one of the arguments is the server service principal name. You can specify the name without including the realm, because `SampleClient` passes the name to such a `createName` method, which appends the default realm as needed.

It is recommended that you always include realms when principal names are used in login configuration files because the behavior of the parsers for such files may be implementation-dependent. They may or may not append the default realm before such names are utilized and subsequent actions may fail if there is no realm in the name.

The Login Configuration File

For this tutorial, we are letting the underlying Kerberos mechanism obtain credentials of the users running `SampleClient` and `SampleServer`, rather than invoking JAAS methods directly (as in the JAAS Authentication tutorial) or indirectly (for example, via the Login utility described

Copyright © 1993, 2026, Oracle and/or its affiliates.

in the Use of JAAS Login Utility tutorial and in the Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges tutorial).

The default Kerberos mechanism implementation supplied by Oracle actually prompts for a Kerberos name and password and authenticates the specified user (or service) to the Kerberos KDC. The mechanism relies on JAAS to perform this authentication.

JAAS supports a pluggable authentication framework, meaning that any type of authentication module can be plugged under a calling application. A login configuration specifies the login module to be used for a particular application. The default JAAS implementation from Oracle requires that the login configuration information be specified in a file. (Note: Some other vendors might not have file-based implementations.) See Appendix B: JAAS Login Configuration File for information as to what a login configuration file is, what it contains, and how to specify which login configuration file should be used.

For this tutorial, the Kerberos login module `com.sun.security.auth.module.Krb5LoginModule` is specified in the configuration file. This login module prompts for a Kerberos name and password and attempts to authenticate to the Kerberos KDC.

Both `SampleClient` and `SampleServer` can use the same login configuration file, if that file contains two entries, one entry for the client side and one for the server side.

The `bcsLogin.conf` login configuration file used for this tutorial is the following:

```
com.sun.security.jgss.initiate  {
  com.sun.security.auth.module.Krb5LoginModule required;
};

com.sun.security.jgss.accept  {
  com.sun.security.auth.module.Krb5LoginModule required storeKey=true
};
```

Entries with these two names (`com.sun.security.jgss.initiate` and `com.sun.security.jgss.accept`) are used by Oracle implementations of GSS-API mechanisms when they need new credentials. Since the mechanism used in this tutorial is the Kerberos V5 mechanism, a Kerberos login module will need to be invoked in order to obtain these credentials. Thus we list `Krb5LoginModule` as a required module in these entries. The `com.sun.security.jgss.initiate` entry specifies the configuration for the client side and the `com.sun.security.jgss.accept` entry for the server side.

The `Krb5LoginModule` succeeds only if the attempt to log in to the Kerberos KDC as a specified entity is successful. When running `SampleClient` or `SampleServer`, the user will be prompted for a name and password.

The `SampleServer` entry **`storeKey=true`** indicates that a secret key should be calculated from the password provided during login and it should be stored in the private credentials of the `Subject` created as a result of login. This key is subsequently utilized during mutual authentication when establishing a security context between `SampleClient` and `SampleServer`.

The `Krb5LoginModule` JavaDoc API documentation describes the configuration options that the `Krb5LoginModule` class supports.

The useSubjectCredsOnly System Property

For this tutorial, we set the system property `javax.security.auth.useSubjectCredsOnly` to `false`, which allows us to relax the usual restriction of requiring a GSS mechanism to obtain necessary credentials from an existing `Subject`, set up by JAAS. When this restriction is

Copyright © 1993, 2026, Oracle and/or its affiliates.

relaxed, it allows the mechanism to obtain credentials from some vendor-specific location. For example, some vendors might choose to use the operating system's cache if one exists, while others might choose to read from a protected file on disk.

When this restriction is relaxed, Oracle's Kerberos mechanism still looks for the credentials in the `Subject` bound to the period of the execution of the current thread, but if it doesn't find any there, it performs JAAS authentication using a Kerberos module to obtain new ones. The Kerberos module prompts you for a Kerberos principal name and password. Note that Kerberos mechanism implementations from other vendors may behave differently when this property is set to `false`. Consult their documentation to determine their implementation's behavior.

Running the SampleClient and SampleServer Programs

To execute the `SampleClient` and `SampleServer` programs, do the following:

- Prepare SampleServer for Execution
- Prepare SampleClient for Execution
- Execute SampleServer
- Execute SampleClient

Prepare SampleServer for Execution

To prepare `SampleServer` for execution, do the following:

1. Copy the following files into a directory accessible by the machine on which you will run `SampleServer`:
   - The `SampleServer.java` source file.
   - The `bcsLogin.conf` login configuration file.
2. Compile `SampleServer.java`:

   ```
   javac SampleServer.java
   ```

Prepare SampleClient for Execution

To prepare `SampleClient` for execution, do the following:

1. Copy the following files into a directory accessible by the machine on which you will run `SampleClient`:
   - The `SampleClient.java` source file.
   - The `bcsLogin.conf` login configuration file.
2. Compile `SampleClient.java`:

   ```
   javac SampleClient.java
   ```

Execute SampleServer

It is important to execute `SampleServer` before `SampleClient` because `SampleClient` will try to make a socket connection to `SampleServer` and that will fail if `SampleServer` is not yet running and accepting socket connections.

Copyright © 1993, 2026, Oracle and/or its affiliates.

To execute `SampleServer`, be sure to run it on the machine it is expected to be run on. This machine name (host name) is specified as an argument to `SampleClient`. The service principal name appears in several places, including the login configuration file and the policy files.

Go to the directory in which you have prepared `SampleServer` for execution. Execute `SampleServer`, specifying

- by `-Djava.security.krb5.realm=<your_realm>` that your Kerberos realm is the one specified.

  For example, if your realm is `KRBNT-OPERATIONS.EXAMPLE.COM` you'd put `-Djava.security.krb5.realm=KRBNT-OPERATIONS.EXAMPLE.COM`.

- by `-Djava.security.krb5.kdc=<your_kdc>` that your Kerberos KDC is the one specified.

  For example, if your KDC is `samplekdc.example.com`you'd put `-Djava.security.krb5.kdc=samplekdc.example.com`.

- by `-Djavax.security.auth.useSubjectCredsOnly=false` that the underlying mechanism can decide how to get credentials. See The useSubjectCredsOnly System Property.

- by `-Djava.security.auth.login.config=bcsLogin.conf` that the login configuration file to be used is `bcsLogin.conf`.

The only argument required by `SampleServer` is one specifying the port number to be used for listening for client connections. Choose a high port number unlikely to be used for anything else. An example would be something like 4444.

The following is the full command to use for Windows, Linux, and macOS.

> **Note**
>
> **Important: In this command, you must replace <*port_number*> with an appropriate port number, <*your_realm*> with your Kerberos realm, and <*your_kdc*> with your Kerberos KDC.**
>
> The `java.security.krb5.kdc` system property interprets the ":" symbol as a separation character for multiple KDCs. If the KDC is not listening on the default port (88), you must provide the default realm and its KDC(s) in a `krb5.conf` file, then set the system property `java.security.krb5.kdc.conf` with the name of this file:
>
> `-Djava.security.krb5.conf=<your_krb5.conf_file>`

Here is the command:

```
java -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djavax.security.auth.useSubjectCredsOnly=false
 -Djava.security.auth.login.config=bcsLogin.conf
 SampleServer <port_number>
```

The full command should appear on one line (or, on Linux or macOS, on multiple lines where each line but the last is terminated with " \" indicating that there is more to come). Multiple lines are used here just for legibility. Since this command is very long, you may need to place it in a .bat file (for Windows) or a .sh file (for Linux or macOS) and then run that file to execute the command.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The `SampleServer` code will listen for socket connections on the specified port. When prompted, type the Kerberos name and password for the service principal. The underlying Kerberos authentication mechanism specified in the login configuration file will log the service principal into Kerberos.

For login troubleshooting suggestions, see Troubleshooting.

Execute SampleClient

To execute `SampleClient`, first go to the directory in which you have prepared `SampleClient` for execution. Execute `SampleClient`, specifying

- by `-Djava.security.krb5.realm=<your_realm>` that your Kerberos realm is the one specified.
- by `-Djava.security.krb5.kdc=<your_kdc>` that your Kerberos KDC is the one specified.
- by `-Djavax.security.auth.useSubjectCredsOnly=false` that the underlying mechanism can decide how to get credentials.
- by `-Djava.security.auth.login.config=bcsLogin.conf` that the login configuration file to be used is `bcsLogin.conf`.

> **Note**
>
> If you use a single equals sign (=) with the `java.security.auth.login.config` system property (instead of a double equals sign (==)), then the configurations specified by both this system property and the `java.security` file are used.

The `SampleClient` arguments are (1) the Kerberos name of the service principal that represents `SampleServer` (see Kerberos User and Service Principal Names, (2) the name of the host (machine) on which `SampleServer` is running, and (3) the port number on which `SampleServer` is listening for client connections.

The following is the full command to use for Windows, Linux, and macOS:

> **Note**
>
> **Important: In this command, you must replace <*service_principal*>, <*host*>, <*port_number*>, <*your_realm*>, and <*your_kdc*> with appropriate values** (and note that the port number must be the same as the port number passed as an argument to `SampleServer`). These values need not be placed in quotes.

Here is the command:

```
java -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djavax.security.auth.useSubjectCredsOnly=false
 -Djava.security.auth.login.config=bcsLogin.conf
 SampleClient <service_principal> <host> <port_number>
```

Type the full command on one line. Multiple lines are used here for legibility. As with the command for executing `SampleServer`, if the command is too long to type directly into your

Copyright © 1993, 2026, Oracle and/or its affiliates.

command window, place it in a .bat file (Windows) or a .sh file (Linux and macOS) and then execute that file.

When prompted, type your Kerberos user name and password. The underlying Kerberos authentication mechanism specified in the login configuration file will log you into Kerberos. The `SampleClient` code requests a socket connection with `SampleServer`. Once `SampleServer` accepts the connection, `SampleClient` and `SampleServer` establish a shared context and then exchange messages as described in this tutorial.

For login troubleshooting suggestions, see Troubleshooting.

JAAS Authentication

This basic tutorial demonstrates how JAAS can be used for the authentication of users: to reliably and securely determine who is currently executing Java code.

JAAS authentication is performed in a *pluggable* fashion. This permits Java applications to remain independent from underlying authentication technologies. New or updated technologies can be plugged in without requiring modifications to the application itself. An implementation for a particular authentication technology to be used is determined at runtime. The implementation is specified in a login configuration file. The authentication technology used for this tutorial is Kerberos. (See Kerberos Requirements.)

The rest of this tutorial consists of the following sections:

1. The Authentication Tutorial Code
2. The Login Configuration
3. Running the Code

If you want to first see the tutorial code in action, you can skip directly to Running the Code and then go back to the other sections to learn about coding and configuration file details.

The Authentication Tutorial Code

Our authentication tutorial code is contained in a single source file, `JaasAcn.java`. This file's `main` method performs the authentication and then reports whether or not authentication succeeded.

The code for authenticating the user is very simple, consisting of just two steps:

1. Instantiating a LoginContext
2. Calling the LoginContext's login Method

Instantiating a LoginContext

In order to authenticate a user, you first need a `javax.security.auth.login.LoginContext`. Here is the basic way to instantiate a `LoginContext`:

```
import javax.security.auth.login.*;
. . .
LoginContext lc =
    new LoginContext(<config file entry name>,
           <CallbackHandler to be used for user interaction>);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

and here is the specific way our tutorial code does the instantiation:

```
import javax.security.auth.login.*;
import com.sun.security.auth.callback.TextCallbackHandler;
. . .
LoginContext lc =
    new LoginContext("JaasSample",
          new TextCallbackHandler());
```

The arguments are the following:

1. **The name of an entry in the JAAS login configuration file**
   This is the name for the `LoginContext` to use to look up an entry for this application in the JAAS login configuration file, described in The Login Configuration. Such an entry specifies the class(es) that implement the desired underlying authentication technology(ies). The class(es) must implement the `LoginModule` interface, which is in the `javax.security.auth.spi` package.

   In our sample code, we use the `Krb5LoginModule` in the `com.sun.security.auth.module` package, which performs Kerberos authentication.

   The entry in the login configuration file we use for this tutorial (see `jaas.conf`) has the name "JaasSample", so that is the name we specify as the first argument to the `LoginContext` constructor.

2. **A `CallbackHandler` instance.**
   When a `LoginModule` needs to communicate with the user, for example to ask for a user name and password, it does not do so directly. That is because there are various ways of communicating with a user, and it is desirable for `LoginModules` to remain independent of the different types of user interaction. Rather, the `LoginModule` invokes a `CallbackHandler` to perform the user interaction and obtain the requested information, such as the user name and password. (`CallbackHandler` is an interface in the `javax.security.auth.callback` package.)

   An instance of the particular `CallbackHandler` to be used is specified as the second argument to the `LoginContext` constructor. The `LoginContext` forwards that instance to the underlying `LoginModule` (in our case `Krb5LoginModule`). An application typically provides its own `CallbackHandler` implementation. A simple `CallbackHandler`, `TextCallbackHandler`, is provided in the `com.sun.security.auth.callback` package to output information to and read input from the command line.

Calling the LoginContext's login Method

Once we have a `LoginContext` `lc`, we can call its `login` method to carry out the authentication process:

```
lc.login();
```

The LoginContext instantiates a new empty `javax.security.auth.Subject` object (which represents the user or service being authenticated; see Subject). The `LoginContext` constructs the configured `LoginModule` (in our case `Krb5LoginModule`) and initializes it with this new `Subject` and `TextCallbackHandler`.

The `LoginContext`'s `login` method then calls methods in the `Krb5LoginModule` to perform the login and authentication. The `Krb5LoginModule` will utilize the `TextCallbackHandler` to obtain the user name and password. Then the `Krb5LoginModule` will use this information

Copyright © 1993, 2026, Oracle and/or its affiliates.

to get the user credentials from the Kerberos KDC. See the Kerberos reference documentation.

If authentication is successful, the `Krb5LoginModule` populates the `Subject` with (1) a Kerberos `Principal` representing the user and (2) the user's credentials (TGT).

The calling application can subsequently retrieve the authenticated `Subject` by calling the `LoginContext`'s `getSubject` method, although doing so is not necessary for this tutorial.

The Login Configuration

JAAS authentication is performed in a pluggable fashion, so applications can remain independent from underlying authentication technologies. A system administrator determines the authentication technologies, or `LoginModules`, to be used for each application and configures them in a login `Configuration`. The source of the configuration information (for example, a file or a database) is up to the current `javax.security.auth.login.Configuration` implementation. The default `Configuration` implementation from Oracle reads configuration information from configuration files, as described in `com.sun.security.auth.login.ConfigFile`.

See Appendix B: JAAS Login Configuration File for information as to what a login configuration file is, what it contains, and how to specify which login configuration file should be used.

The Login Configuration File for This Tutorial

As noted, the login configuration file we use for this tutorial, jass.conf, contains just one entry, which is

```
JaasSample {
  com.sun.security.auth.module.Krb5LoginModule required;
};
```

This entry is named `JaasSample` and that is the name that our tutorial application, `JaasAcn`, uses to refer to this entry. The entry specifies that the `LoginModule` to be used to do the user authentication is the `Krb5LoginModule` in the `com.sun.security.auth.module` package and that this `Krb5LoginModule` is required to "succeed" in order for authentication to be considered successful. The `Krb5LoginModule` succeeds only if the name and password supplied by the user are successfully used to log the user into the Kerberos KDC.

See the `Krb5LoginModule` JavaDoc API documentation for information about all the possible options that can be passed to `Krb5LoginModule`.

Running the Code

To execute our JAAS authentication tutorial code, all you have to do is

1. Place the `JaasAcn.java` application source file and the `jaas.conf` login configuration file into a directory.
2. Compile `JaasAcn.java`:

   ```
   javac JaasAcn.java
   ```

3. Execute the `JaasAcn` application, specifying

Copyright © 1993, 2026, Oracle and/or its affiliates.

- by `-Djava.security.krb5.realm=<your_realm>` that your Kerberos realm is the one specified.

  For example, if your realm is `KRBNT-OPERATIONS.EXAMPLE.COM` you'd put `-Djava.security.krb5.realm=KRBNT-OPERATIONS.EXAMPLE.COM`.
- by `-Djava.security.krb5.kdc=<your_kdc>` that your Kerberos KDC is the one specified.

  For example, if your KDC is `samplekdc.example.com` you'd put `-Djava.security.krb5.kdc=samplekdc.example.com`.
- by `-Djava.security.auth.login.config=jaas.conf` that the login configuration file to be used is `jaas.conf`.

The following is the full command:

> **Note**
>
> **Be sure to replace <*your_realm*> with your Kerberos realm, and <*your_kdc*> with your Kerberos KDC.**

```
java -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djava.security.auth.login.config=jaas.conf JaasAcn
```

Type all that on one line. Multiple lines are used here for legibility.

You will be prompted for your Kerberos user name and password, and the underlying Kerberos authentication mechanism specified in the login configuration file will log you into Kerberos. If your login is successful, you will see the following message:

```
Authentication succeeded!
```

If the login is not successful (for example, if you misspell your password), you will see

```
Authentication failed:
```

followed by a reason for the failure. For example, if you mistype your user name, you may see a message like the following (where the formatting is slightly modified here to increase legibility):

```
Authentication failed:
  Kerberos Authentication Failed:
    javax.security.auth.login.LoginException:
      KrbException: Client not found in Kerberos database
```

For login troubleshooting suggestions, see Troubleshooting Logins.

After fixing any problems, re-run the program to try again.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Use of JAAS Login Utility

The JAAS Authentication tutorial shows how you can use the `LoginContext` and `Subject` classes to write a program to authenticate users to verify their identity.

This tutorial describes a Login utility that performs these operations and then executes any specified application as the authenticated user.

Use of the Login utility with a sample application is demonstrated in this tutorial. The next tutorial, Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges, a client/server application using the Java GSS-API, also uses the Login utility.

It is not necessary to read the previous tutorial on JAAS authentication prior to reading this one. However, you should read Appendix B: JAAS Login Configuration File for information as to what a login configuration file is, since one is needed for this and all other tutorials in this series.

As with all tutorials in this series of tutorials, the underlying technology used to support authentication is Kerberos. See Kerberos Requirements.

- What You Need to Know About the Login Utility
- Application and Other File Requirements
- The Sample Application Program
- The Login Configuration File
- Running the Sample Program with the Login Utility

If you want to first see the tutorial code in action, you can skip directly to Running the Sample Program with the Login Utility and then go back to the other sections to learn more.

What You Need to Know About the Login Utility

You do not need to understand the code contained in `Login.java`; you can just use it as is. However, you need to understand some facts about what it does so that your program and login configuration file will properly work with it. The following is a summary of these facts, followed by sections with further information and examples.

The `Login` class does the following:

- Assumes it is passed, as arguments, your application's top-level class name, followed by any arguments your application may require.
- Assumes that the class name of your top-level application class is also used as the name of the entry to be looked up in your login configuration file.
- Specifies the `TextCallbackHandler` class (from the `com.sun.security.auth.callback` package) as the class to be used when communicating with the user. This class can prompt the user for a user name and password.
- Uses a `LoginContext` to authenticate the user. The `LoginContext` invokes the appropriate authentication technology, or `LoginModule`, to perform the authentication. `LoginModules` use a `CallbackHandler` (in our case, `TextCallbackHandler`) as needed to communicate with the user.
- Allows the user three attempts to successfully log in.
- Creates an instance of the `MyAction` class (also in `Login.java`), passing it the application arguments, if any.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Invokes `Subject.callAs`, passing it a `Subject` representing the user and the `MyAction` instance. The result is that the public static main method from your application is invoked and your application code is considered to be executed on behalf of the user.

Application and Other File Requirements

To utilize the Login utility to authenticate the user and execute your application, you may need a small number of additions or modifications to your login configuration file as described in Login Configuration File Requirements.

Application Requirements

In order to utilize the Login utility, your application code does not need anything special. All you need is for the entry point of your application to be the `main` method of a class you write, as usual.

The way to invoke Login such that it will authenticate the user and then instantiate `MyAction` to invoke your application is the following:

```
java <options> Login <AppName> <app arguments>
```

where <*AppName*> is your application's top-level class name and <*app arguments*> are any arguments required by your application. See Running the Sample Program with the Login Utility for the full command used for this tutorial.

Login Configuration File Requirements

Whenever a LoginContext is used to authenticate the user, you need a login configuration file to specify the desired login module. See the The Login Configuration section in the JAAS authentication tutorial for more information as to what a login configuration file is and what it contains.

When you use the Login utility, the name for the login configuration file entry must be exactly the same as your top-level application class name. See The Login Configuration File in this tutorial for an example.

The Sample Application Program

The `Sample.java` application does the following:

- Reads and prints the value of the `java.home` system property,
- Reads and prints the value of the `user.home` system property, and
- Determines whether or not a file named `foo.txt` exists in the current directory.

The Login Configuration File

The `sample.conf` login configuration file for this tutorial contains a single entry, just like the login configuration file for the JAAS Authentication tutorial. The entry contents are the same since the class implementing the desired authentication technology in both cases is the `Krb5LoginModule` in the `com.sun.security.auth.module` package.

The only difference is the name used for the entry. In the previous tutorial we used the name "JaasSample", since that is the name used by the `JaasAcn` class to look up the entry. When you use the Login utility with your application, it expects the name for your login configuration file entry to be the same as the name of your top-level application class. That application class

Copyright © 1993, 2026, Oracle and/or its affiliates.

for this tutorial is named "Sample" so that must also be the name of the login configuration file entry. Thus the login configuration file looks like the following:

```
Sample {
   com.sun.security.auth.module.Krb5LoginModule required;
};
```

The "required" indicates that login using the `Krb5LoginModule` is required to "succeed" in order for authentication to be considered successful. The `Krb5LoginModule` succeeds only if the name and password supplied by the user are successfully used to log the user into the Kerberos KDC.

See the `Krb5LoginModule` JavaDoc API documentation for information about all the possible options that can be passed to `Krb5LoginModule`.

Running the Sample Program with the Login Utility

To execute the `Sample` application with the Login utility, do the following:

1. Place the following files into a directory:
   - The `Login.java` source file.
   - The `Sample.java` source file.
   - The `sample.conf` login configuration file.
2. Compile `Login.java` and `Sample.java`:

   ```
   javac Login.java Sample.java
   ```

   Note that `Login.java` contains two classes and thus compiling `Login.java` creates `Login.class` and `MyAction.class`.
3. Create a JAR file named `Login.jar` containing `Login.class and MyAction.class`:

   ```
   jar -cvf Login.jar Login.class MyAction.class
   ```

4. Create a JAR file named `Sample.jar` containing `Sample.class`:

   ```
   jar -cvf Sample.jar Sample.class
   ```

5. Execute the `Login` class, specifying
   - An appropriate `-classpath` clause that classes should be searched for in the `Login.jar` and `Sample.jar` JAR files
   - `-Djava.security.krb5.realm=<your_realm>` that your Kerberos realm is the one specified
   - `-Djava.security.krb5.kdc=<your_kdc>` that your Kerberos KDC is the one specified
   - `-Djava.security.auth.login.config=sample.conf` that the login configuration file to be used is `sample.conf`

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> If you use a single equals sign (=) with the `java.security.auth.login.config` system property (instead of a double equals sign (==)), then the configurations specified by both this system property and the `java.security` file are used.

You pass the name of your application (in this case, `Sample`) as an argument to `Login`. You would then add as arguments any arguments required by your application, but in our case `Sample` does not require any.

The following are the full commands to use for Windows, Linux, and macOS. The only difference is that on Windows you use semicolons to separate classpath items, while you use colons for that purpose on Linux and macOS. **Be sure to replace <*your_realm*> with your Kerberos realm, and <*your_kdc*> with your Kerberos KDC.**

Here is the full command for Windows:

```
java -classpath Login.jar;Sample.jar
 -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djava.security.auth.login.config=sample.conf Login Sample
```

Here is the full command for Linux and macOS:

```
java -classpath Login.jar:Sample.jar
 -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djava.security.auth.login.config=sample.conf Login Sample
```

Type the full command on one line. Multiple lines are used here for legibility. If the command is too long for your system, you may need to place it in a `.bat` file (for Windows) or a `.sh` file (for Linux and macOS) and then run that file to execute the command.

You will be prompted for your Kerberos user name and password, and the underlying Kerberos login module specified in the login configuration file will log you into Kerberos. Once authentication is successfully completed, the `Sample` code will be executed on behalf of you, the user. You will see a display of the values of your `java.home` and `user.home` system properties and a statement as to whether or not you have a file named `foo.txt` in the current directory.

For login troubleshooting suggestions, see Troubleshooting Logins.

Use of JAAS Login Utility and Java GSS-API for Secure Message Exchanges

This tutorial presents two sample applications to demonstrate the use of the Java GSS-API. This API permits secure exchanges of messages between communicating applications. Here are the sample client and server applications you'll need for this tutorial:

- `SampleClient.java`
- `SampleServer.java`

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> This tutorial uses the same client and server applications as the Use of Java GSS-API for Secure Message Exchanges Without JAAS Programming tutorial. In that tutorial, JAAS (Java Authentication and Authorization Service) programming is not required. Instead, you let the underlying mechanism decide how to get credentials.

This tutorial uses a more complex login configuration file.

Topics

- Before You Start: Recommended Reading
- Overview of the Client and Server Applications
- Kerberos User and Service Principal Names
- The Login Configuration File
- Running the SampleClient and SampleServer Programs

Before You Start: Recommended Reading

In this Java GSS-API tutorial, the first step is JAAS authentication. Previous tutorials demonstrated the use of JAAS for user authentication and presented examples of login configuration files (specifying the underlying authentication technology to be used) that JAAS requires. Applications in the JAAS introductory tutorial, JAAS Authentication made direct calls to JAAS methods. The Use of JAAS Login Utility tutorial showed the use of a utility program that frees the application from having to do this. The client and server applications in the current tutorial also use the same utility program, so we recommend you **read the login utility tutorial first**.

As with all tutorials in this series, the underlying technology used to support authentication and secure communication for the applications in this tutorial is Kerberos. See Kerberos Requirements.

Overview of the Client and Server Applications

The applications for this tutorial are named `SampleClient` and `SampleServer`.

Each is invoked by executing the Login utility supplied with this tutorial and passing it as arguments the name of the application (`SampleClient` or `SampleServer`), followed by the arguments needed by the application. The Login utility uses a JAAS `LoginContext` to authenticate the user using Kerberos. Finally, the Login utility invokes the `main` method of the application class, in our case either `SampleClient` or `SampleServer`, and passes the application its arguments.

Here is a summary of execution of the `SampleClient` and `SampleServer` applications:

1. Run the `SampleServer` application by running the Login utility and passing it as arguments the name "`SampleServer`" followed by the arguments for the `SampleServer` program. The Login utility prompts you for the password for the principal that `SampleServer` should run as. (See Kerberos User and Service Principal Names.) After authentication is complete, `SampleServer` is run it:
    a. Reads its argument, the port number that it should listen on for client connections.
    b. Creates a `ServerSocket` for listening for client connections on that port.

Copyright © 1993, 2026, Oracle and/or its affiliates.

c. Listens for a connection.

2. Run the `SampleClient` application (possibly on a different machine), by running the Login utility and passing it as arguments the name "`SampleClient`" followed by the arguments for the `SampleClient` program. The Login utility prompts you for your Kerberos name and password. After authentication is complete, `SampleClient` is run. It

   a. Reads its arguments: (1) The name of the Kerberos principal that represents `SampleServer`. (See Kerberos User and Service Principal Names.), (2) the name of the host (machine) on which `SampleServer` is running, and (3) the port number on which `SampleServer` listens for client connections.

   b. Attempts a socket connection with the `SampleServer`, using the host and port it was passed as arguments.

3. The socket connection is accepted by `SampleServer` and both applications initialize a `DataInputStream` and a `DataOutputStream` from the socket input and output streams, to be used for future data exchanges.

4. `SampleClient` and `SampleServer` each instantiate a `GSSContext` and establish a shared context that will enable subsequent secure data exchanges.

5. `SampleClient` and `SampleServer` can now securely exchange messages.

6. When `SampleClient` and `SampleServer` are done exchanging messages, they perform clean-up operations.

> **Note**

Refer to the The SampleClient and SampleServer Code section of the Use of Java GSS-API for Secure Message Exchanges Without JAAS Programming tutorial for a full discussion of the code used in this tutorial.

Kerberos User and Service Principal Names

Since the underlying authentication and secure communication technology used by this tutorial is Kerberos V5, we use Kerberos-style principal names wherever a user or service is called for (see Principals).

For example, when you run `SampleClient` you are asked to provide your **user name**. Your Kerberos-style user name is simply the user name you were assigned for Kerberos authentication. It consists of a base user name (like `mjones`) followed by an "`@`" and your realm (like `mjones@KRBNT-OPERATIONS.EXAMPLE.COM`).

A server program like `SampleServer` is typically considered to offer a "service" and to be run on behalf of a particular "**service principal**." A service principal name for `SampleServer` is needed in several places:

- When you run `SampleServer` you must log in as the appropriate service principal. The login configuration file for this tutorial actually specifies the service principal name (as an option to the `Krb5LoginModule`), so the JAAS authentication (done by the `Login` utility) just asks you to specify the password for that service principal. If you specify the correct password, the authentication is successful, a `Subject` is created containing a `Principal` with the service principal name. The subsequently-executed code (the `SampleServer` code) is considered to be executed on behalf of the specified principal.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- When you run `SampleClient`, one of the arguments is the service principal name. This is needed so `SampleClient` can initiate establishment of a security context with the appropriate service.

Throughout this document, and in the accompanying login configuration file, `service_principal@your_realm` is used as a placeholder to be replaced by the actual name to be used in your environment. *Any* Kerberos principal can actually be used for the service principal name. **So for the purposes of trying out this tutorial, you could use your user name as both the client user name and the service principal name.**

In a production environment, system administrators typically like servers to be run as specific principals only and may assign a particular name to be used. Often the Kerberos-style service principal name assigned is of the form

```
service_name/machine_name@realm;
```

For example, an nfs service run on a machine named "raven" in the realm named `KRBNT-OPERATIONS.EXAMPLE.COM` could have the service principal name

```
nfs/raven@KRBNT-OPERATIONS.EXAMPLE.COM
```

Such multi-component names are not required, however. Single-component names, just like those of user principals, can be used. For example, an installation might use the same ftp service principal `ftp@realm` for all ftp servers in that realm, while another installation might have different ftp principals for different ftp servers, such as `ftp/host1@realm` and `ftp/host2@realm` on machines `host1` and `host2`, respectively.

When the Realm is Required in Principal Names

If the realm of a user or service principal name is the default realm (see Kerberos Requirements), you can leave off the realm when you are logging into Kerberos (that is, when you are prompted for your user name). Thus, for example, if your user name is `mjones@KRBNT-OPERATIONS.EXAMPLE.COM`, and you run `SampleClient`, when it requests your user name you could just specify `mjones`, leaving off the realm. The name is interpreted in the context of being a Kerberos principal name and the default realm is appended, as needed.

You can also leave off the realm if a principal name will be converted to a `GSSName` by a `GSSManager createName` method. For example, when you run `SampleClient`, one of the arguments is the server service principal name. You can specify the name without including the realm, because `SampleClient` passes the name to such a `createName` method, which appends the default realm as needed.

It is recommended that you always include realms when principal names are used in login configuration files because the behavior of the parsers for such files may be implementation dependent. They may or may not append the default realm before such names are utilized and subsequent actions may fail if there is no realm in the name.

The Login Configuration File

Whenever JAAS is used, a login configuration is required to specify the desired authentication technology. (See Appendix B: JAAS Login Configuration File for more information about what a login configuration file is.) Both `SampleClient` and `SampleServer` can use the same login configuration file, if that file contains two entries, one entry for the client side and one for the server side.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The csLogin.conf login configuration file used for this tutorial is the following:

```
SampleClient {
  com.sun.security.auth.module.Krb5LoginModule required;
};

SampleServer {
  com.sun.security.auth.module.Krb5LoginModule required storeKey=true
    principal="service_principal@your_realm";
};
```

Note that the name for each entry matches the respective class names for our two top-level applications, `SampleClient` and `SampleServer`. Recall that this is also the name that is passed to the Login utility that performs JAAS operations for the application. That utility expects the name of the entry to be looked up in your login configuration file to be the same as the name it is passed.

Both entries specify that Oracle's Kerberos V5 LoginModule must be used to successfully authenticate the user. The Krb5LoginModule succeeds only if the attempt to log in to the Kerberos KDC as a specified entity is successful. In the case of `SampleClient`, the user will be prompted for their name and password. In the case of `SampleServer`, a name is already supplied in this login configuration file (the specified principal) and the user running `SampleServer` is just asked for the password for the entity specified by that name. They must specify the correct password in order for authentication to succeed.

The `SampleServer` entry **`storeKey=true`** indicates that a secret key should be calculated from the password provided during login and it should be stored in the private credentials of the `Subject` created as a result of login. This key is subsequently utilized during mutual authentication when establishing a security context between `SampleClient` and `SampleServer`.

The `Krb5LoginModule` has a **`principal`** option that can be used to specify that only the specified principal (entity/user) should be logged in for the given program. Here, the `SampleClient` entry does not specify a principal (although it could, if desired), so the user is prompted for a user name and password and anyone with a valid user name and password can run `SampleClient`. `SampleServer`, on the other hand, indicates a particular principal because system administrators usually like servers to be run as specific principals only. In this case, the user running `SampleServer` is prompted for that principal's password and must supply the correct one in order for authentication to succeed.

Note that you must replace `service_principal@your_realm` with the name of the service principal that represents `SampleServer`. (See Kerberos User and Service Principal Names.)

If the server has a keytab file containing secret keys, then use the following JAAS login entry:

```
SampleServer {
  com.sun.security.auth.module.Krb5LoginModule required
  principal="service_principal@your_realm"
  storeKey=true useKeyTab=true keyTab=keytab.file.name
  isInitiator=false;
  };
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Because the keytab file already provides the keys, you will not be prompted for a password. If the keytab file contains keys for more than one service principal and the server is designed to act as all these service principals, then you can set the principal entry to the following:

```
principal=*
```

See the `Krb5LoginModule` JavaDoc API documentation for information about all the possible options that can be passed to `Krb5LoginModule`.

Running the SampleClient and SampleServer Programs

To execute the `SampleClient` and `SampleServer` programs, do the following:

- Prepare SampleServer for Execution
- Prepare SampleClient for Execution
- Execute SampleServer
- Execute SampleClient

Prepare SampleServer for Execution

To prepare `SampleServer` for execution, do the following:

1. Copy the following files into a directory accessible by the machine on which you will run `SampleServer`:
   - The `Login.java` source file.
   - The `SampleServer.java` source file.
   - The `csLogin.conf` login configuration file.
2. Replace `service_principal@your_realm` in `csLogin.conf` with the name of the service principal representing `SampleServer` (see Kerberos User and Service Principal Names).
3. Compile `Login.java` and `SampleServer.java`:

   ```
   javac Login.java SampleServer.java
   ```

   Note that `Login.java` contains two classes and thus compiling `Login.java` creates `Login.class` and `MyAction.class`.
4. Create a JAR file named `Login.jar` containing `Login.class and MyAction.class`:

   ```
   jar -cvf Login.jar Login.class MyAction.class
   ```

5. Create a JAR file named `SampleServer.jar` containing `SampleServer.class`:

   ```
   jar -cvf SampleServer.jar SampleServer.class
   ```

Prepare SampleClient for Execution

To prepare `SampleClient` for execution, do the following:

1. Copy the following files into a directory accessible by the machine on which you will run `SampleClient`:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- The `Login.java` source file.
- The `SampleClient.java` source file.
- The `csLogin.conf` login configuration file.

2. Compile `Login.java` and `SampleClient.java`:

   ```
   javac Login.java SampleClient.java
   ```

3. Create a JAR file named `Login.jar` containing `Login.class and MyAction.class`:

   ```
   jar -cvf Login.jar Login.class MyAction.class
   ```

4. Create a JAR file named `SampleClient.jar` containing `SampleClient.class`:

   ```
   jar -cvf SampleClient.jar SampleClient.class
   ```

Execute SampleServer

It is important to execute `SampleServer` before `SampleClient` because `SampleClient` will try to make a socket connection to `SampleServer` and that will fail if `SampleServer` is not yet running and accepting socket connections.

To execute `SampleServer`, be sure to run it on the machine it is expected to be run on. This machine name (host name) is specified as an argument to `SampleClient`. The service principal name appears in several places, including the login configuration file.

Go to the directory in which you have prepared `SampleServer` for execution. Execute the `Login` class, specifying

- An appropriate `-classpath` clause that classes should be searched for in the `Login.jar` and `SampleServer.jar` JAR files
- `-Djava.security.krb5.realm=<your_realm>` that your Kerberos realm is the one specified. For example, if your realm is `KRBNT-OPERATIONS.EXAMPLE.COM` you'd put `-Djava.security.krb5.realm=KRBNT-OPERATIONS.EXAMPLE.COM`.
- `-Djava.security.krb5.kdc=<your_kdc>` that your Kerberos KDC is the one specified. For example, if your KDC is `samplekdc.example.com` you'd put `-Djava.security.krb5.kdc=samplekdc.example.com`.
- `-Djava.security.auth.login.config=csLogin.conf` that the login configuration file to be used is `csLogin.conf`

> **Note**
>
> If you use a single equals sign (=) with the `java.security.auth.login.config` system property (instead of a double equals sign (==)), then the configurations specified by both this system property and the `java.security` file are used.

You pass the name of your application (in this case, `SampleServer`) as an argument to Login. You then add as arguments any arguments required by your application, which in the case of `SampleServer` is a single argument specifying the port number to be used for listening for client connections. Choose a high port number unlikely to be used for anything else. An example would be something like 4444.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The following are the full commands to use for Windows, Linux, and macOS. The only difference is that Windows you use semicolons to separate class path items, while you use colons for that purpose on Linux, and macOS.

> **Note**
>
> In these commands, you must replace <*port_number*> with an appropriate port number, <*your_realm*> with your Kerberos realm, and <*your_kdc*> with your Kerberos KDC.

Here is the command for Windows:

```
java -classpath Login.jar;SampleServer.jar
 -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djava.security.auth.login.config=csLogin.conf
 Login SampleServer <port_number>
```

Here is the command for Linux, and macOS:

```
java -classpath Login.jar:SampleServer.jar
 -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djava.security.auth.login.config=csLogin.conf
 Login SampleServer <port_number>
```

Type the full command on one line. Multiple lines are used here for legibility. If the command is too long for your system, you may need to place it in a `.bat` file (for Windows) or a `.sh` file (for Linux and macOS) and then run that file to execute the command.

You will be prompted for the Kerberos password for the service principal. The underlying Kerberos authentication mechanism specified in the login configuration file will log the service principal into Kerberos. Once authentication is successfully completed, the `SampleServer` code will be executed on behalf of the service principal. It will listen for socket connections on the specified port.

For login troubleshooting suggestions, see Troubleshooting Logins.

Execute SampleClient

To execute `SampleClient`, go to the directory in which you have prepared `SampleClient` for execution. Then execute the `Login` class, specifying

- An appropriate `-classpath` clause that classes should be searched for in the `Login.jar` and `SampleClient.jar` JAR files
- `-Djava.security.krb5.realm=`<*your_realm*> that your Kerberos realm is the one specified.
- `-Djava.security.krb5.kdc=`<*your_kdc*> that your Kerberos KDC is the one specified.
- `-Djava.security.auth.login.config=csLogin.conf` that the login configuration file to be used is `csLogin.conf`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Pass to Login the name of your application (`SampleClient`) followed by the arguments required by `SampleClient`. The `SampleClient` arguments are (1) the Kerberos name of the service principal that represents `SampleServer` (see Kerberos User and Service Principal Names, (2) the name of the host (machine) on which `SampleServer` is running, and (3) the port number on which `SampleServer` is listening for client connections.

The following are the full commands to use for Windows, Linux, and macOS.

> **Note**
>
> **Important: In these commands, you must replace <*service_principal*>, <*host*>, <*port_number*>, <*your_realm*>, and <*your_kdc*> with appropriate values** (and note that the port number must be the same as the port number passed as an argument to `SampleServer`). These values need not be placed in quotes.

Here is the command for Windows:

```
java -classpath Login.jar;SampleClient.jar
 -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_kdc>
 -Djava.security.auth.login.config=csLogin.conf
 Login SampleClient <service_principal> <host> <port_number>
```

Here is the command for Linux, and macOS:

```
java -classpath Login.jar:SampleClient.jar
 -Djava.security.krb5.realm=<your_realm>
 -Djava.security.krb5.kdc=<your_realm>
 -Djava.security.auth.login.config=csLogin.conf
 Login SampleClient <service_principal> <host> <port_number>
```

Type the full command on one line. Multiple lines are used here for legibility. As with the command for executing `SampleServer`, if the command is too long to type directly into your command window, place it in a `.bat` file (Windows) or a `.sh` file (Linux and macOS) and then execute that file.

When prompted, type your Kerberos user name and password. The underlying Kerberos authentication mechanism specified in the login configuration file will log you into Kerberos. Once authentication is successfully completed, the `SampleClient` code will be executed on behalf of you. It will request a socket connection with `SampleServer`. Once `SampleServer` accepts the connection, `SampleClient` and `SampleServer` establish a shared context and then exchange messages as described in this tutorial.

For login troubleshooting suggestions, see Troubleshooting Logins.

Using Credentials Delegated from the Client

The most complete type of client impersonation is possible if the client delegates its credentials to the server.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Recall that prior to context establishment with the context acceptor (the server in our previous tutorial), the context initiator (the client) sets various context options. If the initiator calls the `requestCredDeleg` method on the `context` object with a `true` argument, as in

```
context.requestCredDeleg(true);
```

then this requests that the initiator's credentials be delegated to the acceptor during context establishment.

Delegation of credentials from the initiator to the acceptor enables the acceptor to authenticate itself as an agent or delegate of the initiator.

First, after context establishment, the acceptor must determine whether or not credential delegation actually took place. It does so by calling the `getCredDelegState` method:

```
boolean delegated = context.getCredDelegState();
```

If credentials were delegated, the acceptor can obtain those credentials by calling the `getDelegCr` method:

```
GSSCredential clientCr = context.getDelegCred();
```

The resulting `GSSCredential` object can then be used to initiate subsequent GSS-API contexts as a "delegate" of the initiator. For example, the server could authenticate as the client to a backend server that cares more about who the original client was than who the intermediate server is.

Acting as the client, the server can establish a connection with the backend server, establish a joint security context, and exchange messages in basically the same manner that the client and server did.

One way it could be done is that when the server calls the `createContext` method of a `GSSManager`, it could pass `createContext` the delegated credentials instead of passing a `null`.

Constrained Delegation

If constrained delegation is configured in a KDC server, then, on the server side, the `getCredDelegState()` call might still return true and `getDelegCred()` would return delegated credentials, depending on the KDC settings, even if the client has not called `requestCredDeleg(true)`.

Kerberos Requirements

Kerberos Version 5 is used for both the authentication and secure communication aspects of the client and server applications developed in this tutorial. The reader is assumed to already be familiar with Kerberos. See the Kerberos reference documentation.

The JAAS framework, and the Kerberos mechanism required by the Java GSS-API methods, are built into JDKs from all vendors. The Kerberos LoginModule required for the JAAS authentication in this tutorial may not be available in all vendors' JDKs. We will be using the LoginModule for Kerberos provided by Oracle's JDK.

In order to run the sample programs, you will need access to a Kerberos installation. As described in the following sections, you may also need a `krb5.conf` Kerberos configuration file and an indication as to where that file is located.

Copyright © 1993, 2026, Oracle and/or its affiliates.

As with all Kerberos installations, a Kerberos Key Distribution Center (KDC) is required. It needs to contain the user name and password you will use to be authenticated to Kerberos.

> **Note**
>
> A KDC implementation is part of a Kerberos installation and not a part of the JDK.

As with most Kerberos installations, a Kerberos configuration file `krb5.conf` is consulted to determine such things as the default realm and KDC. If you are using a Kerberos implementation that does not include a `krb5.conf` file (such as one from Windows), you will either need to create one or use system properties as described in Setting Properties to Indicate the Default Realm and KDC.

Setting Properties to Indicate the Default Realm and KDC

Typically, the default realm and the KDC for that realm are indicated in the Kerberos `krb5.conf` configuration file. However, if you like, you can instead specify these values by setting the following system properties to indicate the realm and KDC, respectively:

```
java.security.krb5.realm
java.security.krb5.kdc
```

If you set one of these properties you must set them both.

Also note that if you set these properties, then no cross-realm authentication is possible unless a `krb5.conf` file is also provided from which the additional information required for cross-realm authentication may be obtained.

If you set values for these properties, then they override the default realm and KDC values specified in `krb5.conf` (if such a file is found). The `krb5.conf` file is still consulted if values for items other than the default realm and KDC are needed. If no `krb5.conf` file is found, then the default values used for these items are implementation-specific.

Locating the krb5.conf Configuration File

The essential Kerberos configuration information is the default realm and the default KDC. As shown in Setting Properties to Indicate the Default Realm and KDC, if you set properties to indicate these values, they are not obtained from a `krb5.conf` configuration file.

If these properties do not have values set, or if other Kerberos configuration information is needed, an attempt is made to find the required information in a `krb5.conf` file. The algorithm to locate the `krb5.conf` file is the following:

- If the system property `java.security.krb5.conf` is set, its value is assumed to specify the path and file name.
- If that system property value is not set, then the configuration file is looked for in the directory
  - `<java-home>\conf\security` (Windows)
  - `<java-home>/conf/security` (Linux, and macOS)

  Here *<java-home>* refers to the directory where the JDK is installed.
- If the file is still not found, then an attempt is made to locate it as follows:

Copyright © 1993, 2026, Oracle and/or its affiliates.

  - `C:\Windows\krb5.ini` (Windows)
  - `/etc/krb5.conf` (Linux)
  - `~/Library/Preferences/edu.mit.Kerberos`, `/Library/Preferences/edu.mit.Kerberos`, or `/etc/krb5.conf` (macOS)
- If the file is still not found, and the configuration information being searched for is *not* the default realm and KDC, then implementation-specific defaults are used. If, on the other hand, the configuration information being searched for is the default realm and KDC because they weren't specified in system properties, and the `krb5.conf` file is not found either, then an exception is thrown.
- On Windows, if a `krb5.conf` file cannot be found or it does not contain settings for the default realm and its KDC, then the environment variables `USERDNSDOMAIN` and `LOGONSERVER` are used as the default realm and its KDC.

Naming Conventions for Realm Names and Hostnames

By convention, all Kerberos realm names are uppercase and all DNS hostname and domain names are lowercase. On Windows, domains are also Kerberos realms; however, the realm name is always the uppercase version of the domain name.

Hostnames are case insensitive and by convention they are all lowercase. They must resolve to the same hostname on the client and server by their respective naming services.

However, in the Kerberos database hostnames are case sensitive. In all host-based Kerberos service principals in the KDC, hostnames are case-sensitive. The hostnames used in the Kerberos service principal names must exactly match the hostnames returned by the naming service. For example, if the naming service returns a fully qualified lowercased DNS hostname, such as `raven.example.com`, then the administrator must use the same fully qualified lowercased DNS hostname when creating host-based principal names in the KDC: `host/raven.example.com`.

Cross-Realm Authentication

In cross-realm authentication, a principal in one realm can authenticate to principals in another realm.

In Kerberos, cross-realm authentication is implemented by sharing an encryption key between two realms. The KDCs in two different realms share a special cross-realm secret; this secret is used to prove identity when crossing the boundary between realms.

The key that is shared is the Ticket Granting Service principal's key. Here's a typical Ticket Granting Service principal for a single realm:

```
ktbtgt/EXAMPLE.COM@EXAMPLE.COM
```

In cross realm authentication, two principals are created on each participating realm. For two realms, `ENG.EAST.EXAMPLE.COM` and `SALES.WEST.EXAMPLE.COM`, these principals would be:

```
krbtgt/ENG.EAST.EXAMPLE.COM@SALES.WEST.EXAMPLE.COM
krbtgt/SALES.WEST.EXAMPLE.COM@ENG.EAST.EXAMPLE.COM
```

These principals, known as remote Ticket Granting Server principals, must be created on both realms.

Copyright © 1993, 2026, Oracle and/or its affiliates.

For a Windows KDC, the `krbtgt` account is created automatically when a Windows domain is created. This account cannot be deleted and renamed.

Source Code for JAAS and Java GSS-API Tutorials

SampleServer.java

```
import org.ietf.jgss.*;
import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;

/**
 * A sample server application that uses JGSS to do mutual authentication
 * with a client using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the client.
 *
 * Every message exchanged with the client includes a 4-byte application-
 * level header that contains the big-endian integer value for the number
 * of bytes that will follow as part of the JGSS token.
 *
 * The protocol is:
 *    1.  Context establishment loop:
 *         a. client sends init sec context token to server
 *         b. server sends accept sec context token to client
 *         ....
 *    2. client sends a wrap token to the server.
 *    3. server sends a mic token to the client for the application
 *       message that was contained in the wrap token.
 */

public class SampleServer  {

    public static void main(String[] args)
        throws IOException, GSSException {

        // Obtain the command-line arguments and parse the port number
        if (args.length != 1) {
            System.err.println("Usage: java <options> Login SampleServer
<localPort>");
            System.exit(-1);
        }

        int localPort = Integer.parseInt(args[0]);
        ServerSocket ss = new ServerSocket(localPort);
        GSSManager manager = GSSManager.getInstance();

        while (true) {
            System.out.println("Waiting for incoming connection...");
            Socket socket = ss.accept();
            DataInputStream inStream =
                new DataInputStream(socket.getInputStream());
            DataOutputStream outStream =
                new DataOutputStream(socket.getOutputStream());
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
System.out.println("Got connection from client "
                   + socket.getInetAddress());

/*
 * Create a GSSContext to receive the incoming request
 * from the client. Use null for the server credentials
 * passed in. This tells the underlying mechanism
 * to use whatever credentials it has available that
 * can be used to accept this connection.
 */
GSSContext context = manager.createContext((GSSCredential)null);

// Do the context establishment loop
byte[] token = null;
while (!context.isEstablished()) {
    token = new byte[inStream.readInt()];
    System.out.println("Will read input token of size "
                       + token.length
                       + " for processing by acceptSecContext");
    inStream.readFully(token);
    token = context.acceptSecContext(token, 0, token.length);

    // Send a token to the peer if one was generated by
    // acceptSecContext
    if (token != null) {
        System.out.println("Will send token of size "
                           + token.length
                           + " from acceptSecContext.");
        outStream.writeInt(token.length);
        outStream.write(token);
        outStream.flush();
    }
}

System.out.print("Context Established! ");
System.out.println("Client is " + context.getSrcName());
System.out.println("Server is " + context.getTargName());

/*
 * If mutual authentication did not take place, then
 * only the client was authenticated to the
 * server. Otherwise, both client and server were
 * authenticated to each other.
 */
if (context.getMutualAuthState())
    System.out.println("Mutual authentication took place!");

/*
 * Create a MessageProp which unwrap will use to return
 * information such as the Quality-of-Protection that was
 * applied to the wrapped token, whether or not it was
 * encrypted, etc. Since the initial MessageProp values
 * are ignored, just set them to the defaults of 0 and false.
 */
MessageProp prop = new MessageProp(0, false);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                /*
                 * Read the token. This uses the same token byte array
                 * as that used during context establishment.
                 */
                token = new byte[inStream.readInt()];
                System.out.println("Will read token of size "
                                   + token.length);
                inStream.readFully(token);
                byte[] bytes = context.unwrap(token, 0, token.length, prop);
                String str = new String(bytes);
                System.out.println("Received data \""
                                   + str + "\" of length " + str.length());
                System.out.println("Confidentiality applied: "
                                   + prop.getPrivacy());

                /*
                 * Now generate a MIC and send it to the client. This is
                 * just for illustration purposes. The integrity of the
                 * incoming wrapped message is guaranteed irrespective of
                 * the confidentiality (encryption) that was used.
                 */

                /*
                 * First reset the QOP of the MessageProp to 0
                 * to ensure the default Quality-of-Protection
                 * is applied.
                 */
                prop.setQOP(0);
                token = context.getMIC(bytes, 0, bytes.length, prop);
                System.out.println("Will send MIC token of size "
                                   + token.length);
                outStream.writeInt(token.length);
                outStream.write(token);
                outStream.flush();
                System.out.println("Closing connection with client "
                                   + socket.getInetAddress());
                context.dispose();
                socket.close();
            }
        }
}
```

bcsLogin.conf

```
/**
 * Login Configuration for JAAS.
 */

com.sun.security.jgss.initiate {
  com.sun.security.auth.module.Krb5LoginModule required;
};

com.sun.security.jgss.accept {
  com.sun.security.auth.module.Krb5LoginModule required storeKey=true;
};
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

SampleClient.java

```
import org.ietf.jgss.*;
import java.net.Socket;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.DataOutputStream;

/**
 * A sample client application that uses JGSS to do mutual authentication
 * with a server using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the server.
 *
 * Every message sent to the server includes a 4-byte application-level
 * header that contains the big-endian integer value for the number
 * of bytes that will follow as part of the JGSS token.
 *
 * The protocol is:
 *    1.  Context establishment loop:
 *         a. client sends init sec context token to server
 *         b. server sends accept sec context token to client
 *         ....
 *    2. client sends a wrap token to the server.
 *    3. server sends a MIC token to the client for the application
 *       message that was contained in the wrap token.
 */

public class SampleClient {

    public static void main(String[] args)
       throws IOException, GSSException  {

        // Obtain the command-line arguments and parse the port number

        if (args.length < 3) {
            System.err.println("Usage: java <options> Login SampleClient "
                               + " <server> <hostName> <port>");
            System.exit(-1);
        }

        String server = args[0];
        String hostName = args[1];
        int port = Integer.parseInt(args[2]);

        Socket socket = new Socket(hostName, port);
        DataInputStream inStream =
          new DataInputStream(socket.getInputStream());
        DataOutputStream outStream =
          new DataOutputStream(socket.getOutputStream());

        System.out.println("Connected to server "
                           + socket.getInetAddress());

        /*
         * This Oid is used to represent the Kerberos version 5 GSS-API
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 * mechanism. It is defined in RFC 1964. We will use this Oid
 * whenever we need to indicate to the GSS-API that it must
 * use Kerberos for some purpose.
 */
Oid krb5Oid = new Oid("1.2.840.113554.1.2.2");

GSSManager manager = GSSManager.getInstance();

/*
 * Create a GSSName out of the server's name. The null
 * indicates that this application does not wish to make
 * any claims about the syntax of this name and that the
 * underlying mechanism should try to parse it as per whatever
 * default syntax it chooses.
 */
GSSName serverName = manager.createName(server, null);

/*
 * Create a GSSContext for mutual authentication with the
 * server.
 *    - serverName is the GSSName that represents the server.
 *    - krb5Oid is the Oid that represents the mechanism to
 *      use. The client chooses the mechanism to use.
 *    - null is passed in for client credentials
 *    - DEFAULT_LIFETIME lets the mechanism decide how long the
 *      context can remain valid.
 * Note: Passing in null for the credentials asks GSS-API to
 * use the default credentials. This means that the mechanism
 * will look among the credentials stored in the current Subject
 * to find the right kind of credentials that it needs.
 */
GSSContext context = manager.createContext(serverName,
                                krb5Oid,
                                null,
                                GSSContext.DEFAULT_LIFETIME);

// Set the desired optional features on the context. The client
// chooses these options.

context.requestMutualAuth(true);  // Mutual authentication
context.requestConf(true);  // Will use confidentiality later
context.requestInteg(true); // Will use integrity later

// Do the context eastablishment loop

byte[] token = new byte[0];

while (!context.isEstablished()) {

    // token is ignored on the first call
    token = context.initSecContext(token, 0, token.length);

    // Send a token to the server if one was generated by
    // initSecContext
    if (token != null) {
        System.out.println("Will send token of size "
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                            + token.length
                            + " from initSecContext.");
                outStream.writeInt(token.length);
                outStream.write(token);
                outStream.flush();
            }

            // If the client is done with context establishment
            // then there will be no more tokens to read in this loop
            if (!context.isEstablished()) {
                token = new byte[inStream.readInt()];
                System.out.println("Will read input token of size "
                            + token.length
                            + " for processing by initSecContext");
                inStream.readFully(token);
            }
        }

        System.out.println("Context Established! ");
        System.out.println("Client is " + context.getSrcName());
        System.out.println("Server is " + context.getTargName());

        /*
         * If mutual authentication did not take place, then only the
         * client was authenticated to the server. Otherwise, both
         * client and server were authenticated to each other.
         */
        if (context.getMutualAuthState())
            System.out.println("Mutual authentication took place!");

        byte[] messageBytes = "Hello There!\0".getBytes();

        /*
         * The first MessageProp argument is 0 to request
         * the default Quality-of-Protection.
         * The second argument is true to request
         * privacy (encryption of the message).
         */
        MessageProp prop =  new MessageProp(0, true);

        /*
         * Encrypt the data and send it across. Integrity protection
         * is always applied, irrespective of confidentiality
         * (i.e., encryption).
         * You can use the same token (byte array) as that used when
         * establishing the context.
         */

        token = context.wrap(messageBytes, 0, messageBytes.length, prop);
        System.out.println("Will send wrap token of size " + token.length);
        outStream.writeInt(token.length);
        outStream.write(token);
        outStream.flush();

        /*
         * Now we will allow the server to decrypt the message,
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
         * calculate a MIC on the decrypted message and send it back
         * to us for verification. This is unnecessary, but done here
         * for illustration.
         */

        token = new byte[inStream.readInt()];
        System.out.println("Will read token of size " + token.length);
        inStream.readFully(token);
        context.verifyMIC(token, 0, token.length,
                          messageBytes, 0, messageBytes.length,
                          prop);

        System.out.println("Verified received MIC for message.");

        System.out.println("Exiting...");
        context.dispose();
        socket.close();
    }
}
```

JaasAcn.java

```
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import com.sun.security.auth.callback.TextCallbackHandler;

/**
 * This JaasAcn application attempts to authenticate a user
 * and reports whether or not the authentication was successful.
 */
public class JaasAcn {

    public static void main(String[] args) {

        // Obtain a LoginContext, needed for authentication. Tell it
        // to use the LoginModule implementation specified by the
        // entry named "JaasSample" in the JAAS login configuration
        // file and to also use the specified CallbackHandler.
        LoginContext lc = null;
        try {
            lc = new LoginContext("JaasSample", new TextCallbackHandler());
        } catch (LoginException le) {
            System.err.println("Cannot create LoginContext. "
                + le.getMessage());
            System.exit(-1);
        }

        try {

            // attempt authentication
            lc.login();

        } catch (LoginException le) {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            System.err.println("Authentication failed:");
            System.err.println("  " + le.getMessage());
            System.exit(-1);

        }

        System.out.println("Authentication succeeded!");

    }
}
```

jass.conf

```
/** Login Configuration for the JaasAcn application
 **/

JaasSample {
   com.sun.security.auth.module.Krb5LoginModule required;
};
```

Login.java

```
import java.io.*;
import java.lang.reflect.*;
import java.util.Arrays;
import java.util.concurrent.*;

import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.Subject;
import com.sun.security.auth.callback.TextCallbackHandler;

/**
 * <p>This class authenticates a <code>Subject</code> and then
 * executes a specified application as that
 * <code>Subject</code>. To use this class, the java interpreter
 * would typically be invoked as:</p>
 *
 * <pre>% java Login <i>applicationclass</i> <i>applicationClass_args</i></
pre>
 *
 * <p><i>applicationClass</i> represents the application to be
 * executed as the authenticated <code>Subject</code>,
 * and <i>applicationClass_args</i> are passed as arguments to
 * <i>applicationClass</i>.</p>
 *
 * <p>To perform the authentication, <code>Login</code> uses a
 * <code>LoginContext</code>. A <code>LoginContext</code> relies
 * on a <code>Configuration</code> to determine the modules that
 * should be used to perform the actual authentication. The
 * location of the <code>Configuration</code> is dependent upon
 * each <code>Configuration</code> implementation.<p>
 *
 * <p>The default <code>Configuration</code> implementation
 * (<code>com.sun.security.auth.login.ConfigFile</code>)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 * allows the <code>Configuration</code> location to be
 * specified (among other ways) through the
 * <code>java.security.auth.login.config</code> system property.
 * Therefore, the <code>Login</code> class can also be invoked
 * as:</p>
 *
 * <pre>% java -Djava.security.auth.login.config=<i>configuration_url</i> \
 *        Login \
 *        <i>your_application_class</i> <i>your_application_class_args</i></
pre>
 */

public class Login {

    /**
     * <p>Instantate a <code>LoginContext</code> using the
     * provided application classname as the index for the login
     * <code>Configuration</code>. Authenticate the
     * <code>Subject</code> (three retries are allowed) and
     * invoke <code>Subject.callAs</code> with the
     * authenticated <code>Subject</code> and a
     * <code>Callable</code>. The <code>Callable</code>
     * loads the provided application class, and then invokes
     * its public static <code>main</code> method, passing it
     * the application arguments.</p>
     *
     * @param args the arguments for <code>Login</code>. The
     *          first argument must be the class name of the
     *          application to be invoked once authentication
     *          has completed, and the subsequent arguments are
     *          the arguments to be passed to that application's
     *          public static <code>main</code> method.
     */
    public static void main(String[] args) {

        // check for the application's main class
        if (args == null || args.length == 0) {
            System.err.println("Invalid arguments: " +
                "Did not provide name of application class.");
            System.exit(-1);
        }

        LoginContext lc = null;
        try {
            lc = new LoginContext(args[0], new TextCallbackHandler());
        } catch (LoginException le) {
            System.err.println("Cannot create LoginContext. "
                + le.getMessage());
            System.exit(-1);
        }

        // the user has 3 attempts to authenticate successfully
        int i;
        for (i = 0; i < 3; i++) {
            try {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                    // attempt authentication
                    lc.login();

                    // if we return with no exception, authentication succeeded
                    break;

                } catch (AccountExpiredException aee) {

                    System.err.println("Your account has expired.  " +
                                       "Please notify your administrator.");
                    System.exit(-1);

                } catch (CredentialExpiredException cee) {

                    System.err.println("Your credentials have expired.");
                    System.exit(-1);

                } catch (FailedLoginException fle) {

                    System.err.println("Authentication Failed");
                    try {
                          Thread.currentThread().sleep(3000);
                    } catch (Exception e) {
                          // ignore
                    }

                } catch (Exception e) {

                    System.err.println("Unexpected Exception - unable to
continue");
                    e.printStackTrace();
                    System.exit(-1);
                }
            }

            // did they fail three times?
            if (i == 3) {
                System.err.println("Sorry");
                System.exit(-1);
            }

            // Run the action as the subject
            try {
                Subject.callAs(lc.getSubject(),
                               new MyAction(args));
            } catch (CompletionException ce) {
                ce.printStackTrace();
                System.exit(-1);
            }

            System.exit(0);
        }
    }

    class MyAction implements Callable<Void> {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    String[] origArgs;

    public MyAction(String[] origArgs) {
        this.origArgs = (String[])origArgs.clone();
    }

    public Void call() throws Exception {

        // get the ContextClassLoader
        ClassLoader cl = Thread.currentThread().getContextClassLoader();

        // successful completion
        return null;
    }
}
```

Sample.java

```
import java.io.File;

public class Sample {

    /**
     * This sample class performs the following operations:
     * <ul>
     * <li> Access the System property <i>java.home</i>
     * <li> Access the System property <i>user.home</i>
     * <li> Access the file <i>foo.txt</i>
     * </ul>
     *
     */
    public static void main (String[] args) {

        // If there were any arguments to read, we'd do it here.

        System.out.println("\nYour java.home property value is: "
                           +System.getProperty("java.home"));

        System.out.println("\nYour user.home property value is: "
                           +System.getProperty("user.home"));

        File f = new File("foo.txt");
        System.out.print("\nfoo.txt does ");
        if (!f.exists())
            System.out.print("not ");
        System.out.println("exist in the current working directory.");
    }
}
```

sample.conf

```
/** Login Configuration for the Sample Application **/

Sample {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
   com.sun.security.auth.module.Krb5LoginModule required;
};
```

csLogin.conf

```
/**
 * Login Configuration for JAAS.
 */

SampleClient {
  com.sun.security.auth.module.Krb5LoginModule required;
};

SampleServer {
  com.sun.security.auth.module.Krb5LoginModule required storeKey=true
principal="service_principal@your_realm";
};
```

Accessing Native GSS-API

To help Java platform applications achieve seamless integration with native applications, the JDK enhances the Java GSS-API to use native GSS-API instead of its own implementation of cryptographic mechanisms when configured to do so. When using the native GSS-API and its underlying native cryptographic mechanisms, the native credentials and settings in users' environment will be picked up automatically. This is different from the default case in which the Java GSS-API uses its own implementation of cryptographic mechanisms. When using Kerberos, Java applications have to supply Kerberos configuration information using the designated Kerberos system properties for the Java GSS-API to function. Introduction to JAAS and Java GSS-API Tutorials covers the default case in great detail, so this section will focus on how to enable or configure Java GSS-API to use native GSS-API.

Before you enable Java GSS-API to use native GSS-API, ensure that native GSS-API and its underlying cryptographic mechanism are available and functioning with user settings. For example, ensure that native GSS libraries are installed at the appropriate directories with proper configurations, and the same applies to the Kerberos library and configurations. Note that native GSS-API assumes that before an application calls its APIs, it has already obtained and stored the mechanism-specific credentials in a location that the native mechanism implementation is aware of. Thus, when an application uses native GSS-API with Kerberos, it must already have obtained the appropriate native credentials, such as Kerberos tickets and keys by using the `kinit` tool on the initiator side, a keytab file on the acceptor side, or default credentials acquired during the system login.

To make the Java GSS-API use native GSS-API, Java applications must explicitly enable this behavior by setting one or more of the following system properties:

- `sun.security.jgss.native` (required): Set this to `true` to enable the Java GSS-API to use the native GSS library.
- `sun.security.jgss.lib` (optional): Set to the full path of the native GSS library. If this is set, then the Java GSS-API looks for the specified library using the default Java library path. If this is not set, then the Java GSS-API uses a default native GSS library. On Windows, this is `sspi.dll`, which is included in the JDK. This library is client-side only and uses the default credentials. On other platforms, the Java GSS-API searches for an existing well-known native GSS library, for example, `libgssapi.so` or `libgssapi_krb5.so` on Linux or `libgssapi_krb5.dylib` on macOS.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> To enable the native GSS-API debugging mode, set the system property `sun.security.nativegss.debug` to `true`. This enables you to follow a program's native GSS-API calls.

As mentioned previously, native GSS-API requires that the application had obtained these credentials and that they are accessible. Java applications can access these native credentials through the Java GSS-API and use them for establishing GSS-API security contexts with peers. Note that when a `Subject` is present, for example,

```
javax.security.auth.Subject.current() != null
```

then the Java GSS-API mandates that the credentials be obtained from the private or public credential sets of the current `Subject` and that the Java GSS-API call must fail if the desired credential cannot be found. Thus, Java platform applications that execute the Java GSS-API calls inside a `Subject.callAs(Subject, Callable<T>)` call should either populate the `Subject`'s credential sets with the appropriate Java `GSSCredential` objects that encapsulate the native credentials or explicitly set the system property `javax.security.auth.useSubjectCredsOnly` to `false` so that the Java GSS-API can obtain credentials from other locations, for example, from native credential caches, in addition to the `Subject`'s credential sets.

When delegated to establish a GSS-API security context on behalf of others, Java applications can either specify the delegated credential, as returned by `GSSContext.getDelegCred()`, explicitly in Java GSS-API calls, or create a `Subject` object with this delegated credential and execute the Java GSS-API calls inside the `Subject.callAs(Subject, Callable<T>)` calls.

Once the native GSS-API is enabled, Java platform applications that indirectly call Java GSS-API through mechanisms or protocols such as Simple Authentication and Security Layer (SASL) (see Java SASL API Programming and Deployment Guide) will also use user's native settings and credentials.

Here is some sample code that helps demonstrate how to use Java GSS-API to establish GSS-API security contexts and securely exchange data between three parties: `SampleClient` contacts `FooServer`, which in turn contacts `FooServer2` on behalf of `SampleClient`. Note:

- The sample code should be invoked with native GSS-API enabled. The Principal names `host@foo.sample.com` and `host@foo2.sample.com` are placeholders and should be replaced with actual principal names in your Kerberos database.
- To simplify the example, token exchanges between peers are represented by two pseudo-methods: `SEND_TOKEN(byte[])` and `READ_TOKEN()`. Their actual implementation are application-specific and thus not shown here.
- To reduce code duplication, context establishment code is referred by a pseudo-method, `ESTABLISH_CONTEXT(GSSContext)`, in the code segments for `SampleClient`, `FooServer`, and `FooServer2`.

The following is the implementation for `ESTABLISH_CONTEXT(GSSContext)` using Java GSS-API.

```
/**
 * ESTABLISH_CONTEXT(GSSContext ctxt): establishes a context
 * with data confidentiality and mutual authentication.
 */
ctxt.requestConf(true);
ctxt.requestMutualAuth(true);

byte[] inToken = new byte[0];
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
byte[] outToken = null;

if (ctxt.isInitiator()) {
    while (!ctxt.isEstablished()) {
        // Note: initSecContext(...) always ignores the arguments
        // for the first call because there is no incoming token.
        outToken = ctxt.initSecContext(inToken, 0, inToken.length);

        // Send the output token if generated.
        if (outToken != null) SEND_TOKEN(outToken); // to acceptor

        // Check whether more incoming tokens are expected.
        if (!ctxt.isEstablished()) {
            inToken = READ_TOKEN(); // from acceptor
        }
    }
} else {
    while (!ctxt.isEstablished()) {
        inToken = READ_TOKEN(); // from initiator
        outToken =
            ctxt.acceptSecContext(inToken, 0, inToken.length);

        // Send the output token if generated.
        if (outToken != null) SEND_TOKEN(outToken); // to initiator
    }
}
```

Following are the code segments for `SampleClient`, `FooServer`, and `FooServer2`:

`SampleClient`: It contacts `FooServer` and delegates the server to act on its behalf. If all goes well, it should get back a personalized hello message produced by `FooServer2`.

```
GSSManager gssMgr = GSSManager.getInstance();
GSSName serverName = gssMgr.createName(
    "host@foo.sample.com", GSSName.NT_HOSTBASED_SERVICE);
GSSContext context = gssMgr.createContext(
    serverName, null /* default mechanism, which is Kerberos*/,
    null /* default initiator cred */,
    GSSContext.DEFAULT_LIFETIME);
context.requestCredDelegState(true);

ESTABLISH_CONTEXT(context);

// Make sure credential delegation is available.
if (!context.getCredDeleg()) {
    context.dispose();
    throw new Exception("credential delegation is denied");
}

byte[] token = READ_TOKEN(); // from "FooServer"
byte[] data =
   context.unwrap(token, 0, token.length, new MessageProp(true));
context.dispose();

// Should print "Hello from FooServer2 to <client name>" where
// <client name> is the name of the default initiator.
System.out.println(new String(data));
```

`FooServer`: It contacts `FooServer2` as `SampleClient` and forwards the received reply to `SampleClient`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

```
GSSManager gssMgr = GSSManager.getInstance();
GSSName myName = gssMgr.createName(
    "host@foo.sample.com", GSSName.NT_HOSTBASED_SERVICE);
GSSCredential myCred = gssMgr.createCredential(
    acceptorName, GSSCredential.INDEFINITE_LIFETIME,
    (Oid[]) null /* default set of mechanisms */,
    GSSCredential.ACCEPT_ONLY);
GSSContext acontext = gssMgr.createContext(myCred);

ESTABLISH_ACC_CONTEXT(acontext);

GSSCredential delegCred = acontext.getDelegCred();
if (delegCred != null) {
    byte[] data, token;
    // Establish a context on client's behalf using the delegated
    // credential.
    GSSName serverName = gssMgr.createName(
        "host@foo2.sample.com", GSSName.NT_HOSTBASED_SERVICE);
    GSSContext icontext = gssMgr.createContext(
        serverName, null /* default mechanism Kerberos */,
        delegCred /* act on SampleClient's behalf */,
        GSSContext.DEFAULT_LIFETIME);

    ESTABLISH_CONTEXT(icontext);

    token = READ_TOKEN(); // from "FooServer2"

    MessageProp msgProp = new MessageProp(true);

    // Forward the reply from FooServer2 to SampleClient.
    data = icontext.unwrap(token, 0, token.length, msgProp);
    token = acontext.wrap(data, 0, data.length, msgProp);
    SEND_TOKEN(token); // to "SampleClient"
    icontext.dispose();
}
acontext.dispose();
```

`FooServer2`: It always replies with a hello message personalized to the name of the initiator of the established context.

```
GSSManager gssMgr = GSSManager.getInstance();
GSSName myName = gssMgr.createName(
    "host@foo2.sample.com", GSSName.NT_HOSTBASED_SERVICE);
GSSCredential myCred = gssMgr.createCredential(
    myName, GSSCredential.INDEFINITE_LIFETIME,
    (Oid[]) null /* default set of mechanisms */,
    GSSCredential.ACCEPT_ONLY);
GSSContext context = gssMgr.createContext(myCred);

ESTABLISH_CONTEXT(context);

byte[] data = new String("Hello from FooServer2 to " +
    context.getSrcName()).getBytes();
byte[] token =
    context.wrap(data, 0, data.length, new MessageProp(true));

SEND_TOKEN(token); // to "FooServer"

context.dispose();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Advanced Security Programming in Java SE Authentication, Secure Communication and Single Sign-On

Java SE offers a rich set of APIs and features for developing secure Java applications and services. The exercise sessions listed here can help you to use the Java SE GSS APIs to build applications that authenticate their users, to communicate securely with other applications and services, and help you to configure your applications in a Kerberos environment to achieve Single Sign-On. In addition, you will also learn how to use stronger encryption algorithms in a Kerberos environment, and how to use Java GSS mechanisms such as SPNEGO to secure the association.

Setting up your Development Environment

Set up your development environment as follows before proceeding to the first exercise:

1. Configure a Kerberos server with accounts used by the exercises. See Appendix A: Setting up Kerberos Accounts.
2. Set up the Key Distribution Center (KDC) and start the Kerberos server.
3. Set up the Kerberos configuration on your client computer.
4. Set up the JDK environment:
   - Set up the `JAVA_HOME` environment variable to point to the JDK installation directory
   - Place `%JAVA_HOME%\bin` (Windows) or `$JAVA_HOME/bin` (Linux or macOS) in the `PATH` environment variable.

Exercises

This session includes six lessons. Each part contains one or more coding exercises. Work through the exercises in sequence:

- Part I : Secure Authentication using the Java Authentication and Authorization Service (JAAS)
  - Exercise 1: Using the JAAS API
  - Exercise 2: Configuring JAAS for Kerberos Authentication
- Part II : Secure Communications using the Java SE Security API
  - Exercise 3: Using the Java Generic Security Service (GSS) API
  - Exercise 4: Using the Java SASL API
- Part III : Deploying for Single Sign-On in a Kerberos Environment
  - Exercise 5: Deploying for Single Sign-On
- Part IV : Secure Communications Using Stronger Encryption Algorithms
  - Exercise 6: Configuring to Use Stronger Encryption Algorithms in a Kerberos Environment, to Secure the Communication
- Part V : Secure Authentication Using SPNEGO Java GSS Mechanism
  - Exercise 7: Using the Java Generic Security Services (GSS) API with SPNEGO
- Part VI: HTTP/SPNEGO Authentication
  - Exercise 8: Using HTTP/SPNEGO Authentication

Copyright © 1993, 2026, Oracle and/or its affiliates.

Part I : Secure Authentication using the Java Authentication and Authorization Service (JAAS)

Exercise 1: Using the JAAS API

Goal of This Exercise

The goal of this exercise is to learn how to use the Java Authentication and Authorization (JAAS) API to perform authentication.

Background for This Exercise

JAAS provides a standard pluggable authentication framework (PAM) for the Java platform. An application uses the JAAS API to perform *authentication* - the process of verifying the identity of the user who is using the application and gathering his identity information into a container called a *subject*. The application can then use the identity information in the subject along with the JAAS API to make *authorization* decisions, to decide whether the authenticated user is allowed to access protected resources or perform restricted actions. This exercise demonstrates JAAS Authentication. It does not demonstrate JAAS Authorization.

Resources for This Exercise

- Java Authentication and Authorization Service (JAAS) Reference Guide
- JAAS Authentication Tutorial
- JAAS JavaDoc API documentation
  - `javax.security.auth`
  - `javax.security.auth.callback`
  - `javax.security.auth.kerberos`
  - `javax.security.auth.login`
  - `javax.security.auth.spi`
  - `javax.security.auth.x500`

Steps to Follow

- Read the `Jass.java` sample code. The code performs the following tasks:
  1. Define a callback handler or use a predefined one.
  2. Create a `LoginContext` with a name that identifies which JAAS configuration entry to use.
  3. Perform the authentication.
  4. Define the task that the authenticated user is to perform.
  5. Perform the action as the authenticated user.
  6. Log out.

  `Subject.callAs` will run the code defined in `MyAction` as the authenticated user. This enables code in `MyAction` to obtain any required identity information for authentication to a service from the subject.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Make sure that the `%JAVA_HOME%/bin` is in the PATH environment variable.
- Compile the modified sample code. You will run this code in subsequent exercises after doing some set up. That ends this exercise.

Summary

This exercise introduced the main classes of the JAAS APIs: `LoginContext` and `Subject`. You learned how to use `LoginContext` to authenticate a user and collect its identity information in a Subject. You then learned how to use the Subject to perform an action as the authenticated user.

Next Steps

Proceed to Exercise 2: Configuring JAAS for Kerberos Authentication to learn how to configure the sample application to use Kerberos for authentication.

Exercise 2: Configuring JAAS for Kerberos Authentication

Goal of This Exercise

The goal of this exercise is to learn how to configure a JAAS application to use Kerberos for authentication.

Kerberos Background for This Exercise

Kerberos is an Internet standard protocol for trusted-third party authentication defined in RFC 4120. It is available on most modern computing platforms today, including Windows and Linux.

The Kerberos architecture is centered around a trusted authentication service called the key distribution center, or KDC. Users and services in a Kerberos environment are referred to as principals; each principal shares a secret (such as a password) with the KDC. A principal authenticates to Kerberos by proving to the KDC that it knows the shared secret. If the authentication is successful, the KDC issues a ticket-granting-ticket (TGT) to the principal. When the principal subsequently wants to authenticate to a service on the network, such as a directory service or a file service, (thereby, acting as a "client" of the service), it gives the TGT to the KDC to obtain a service ticket to communicate with the service. Not only does the service ticket indicate the identities of the client and service principals, it also contains a session key that can be used by the client and service to subsequently establish secure communication. To authenticate to the service, the client sends the service ticket to the service. When the service receives the ticket, it decodes it using the secret it shares with the KDC.

In this architecture, a principal only authenticates directly (once) to the KDC. It authenticates indirectly to all other services via the use of service tickets. Service tickets are how the KDC vouches for the identity of a principal. The ability of a principal to access multiple secure services by performing explicit authentication only once is called single sign-on.

JAAS Background for This exercise

In JAAS, for a client principal, "logging into Kerberos" means acquiring the TGT and placing it in the Subject, so that it can be used for authentication with services that the client will access. For a service principal, "logging into Kerberos" means obtaining the secret keys that the service needs to decode incoming client authentication requests.

Resources for This Exercise

- Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide
- The Kerberos Network Authentication Service (v5)

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Appendix B: JAAS Login Configuration File
- Login module package JavaDoc API documentation:`com.sun.security.auth.module`
- Introduction to JAAS and Java GSS-API Tutorials

Steps to Follow

1. Examine the `jaas-krb5.conf` configuration file.

   This file contains two entries, one named *client* and one named *server*. The client entry indicates that the `LoginContext` must use the **`com.sun.security.auth.module.Krb5LoginModule`**. The server entry indicates that the `LoginContext` must use the same login module, and use keys from the `sample.keytab` file for the principal host/machineName.
2. Determine the hostname of your machine by executing the hostname command.
3. Edit this file and change the entry for server principal to use the name of your machine. For example, if your machine name is j1hol-001, this line in the configuration file should look like this:

   ```
   principal="host/j1hol-001"
   ```

4. Perform client authentication by typing the following command:

   ```
   % java -Djava.security.auth.login.config=jaas-krb5.conf Jaas client
   ```

   You will be prompted for a password. You should see the following output. Replace *password* with a password that is secure.

   ```
   Kerberos password for test: password
   Authenticated principal: [test@J1LABS.EXAMPLE.COM]
   Performing secure action...
   ```

5. Perform server authentication by typing the following command:

   ```
   % java -Djava.security.auth.login.config=jaas-krb5.conf Jaas server
   ```

   You should see the following output:

   ```
   Authenticated principal: [host/j1hol-001@J1LABS.EXAMPLE.COM] Performing
   secure action...
   ```

Summary

In this exercise, you learned how to configure a JAAS application to use a Kerberos login module, both as a client principal who enters his/her username/password interactively, and as a service principal who gets its keys from a `keytab` file.

Next Steps

Proceed to Part II : Secure Communications using the Java SE Security API to learn how to establish secure communication channels using Java security APIs.

Part II : Secure Communications using the Java SE Security API

Copyright © 1993, 2026, Oracle and/or its affiliates.

This part shows you how to build applications that perform secure communications. The Java SE platform provides three standard APIs that allow applications to perform secure communications: The Java Generic Security Service (GSS), the Java SASL API, and the Java Secure Socket Extension (JSSE). When building an application, which of these APIs should you use? The answer depends on many factors, including requirements of the protocol or service, deployment infrastructure, and integration with other security services. For example, if you are building an LDAP client library, you would need to use the Java SASL API because use of SASL is part of LDAP's protocol definition. As an other example, if the service supports SSL, then the client application attempting to access the service would need to use JSSE.

Exercise 3: Using the Java Generic Security Service (GSS) API

Goal of This Exercise

The goal of this exercise is to learn how to use the Java GSS API to perform secure authentication and communication.

Background for This Exercise

The Generic Security Service API provides a uniform C-language interface to access various security services, such as authentication, message integrity, and message confidentiality. The Java GSS API provides the corresponding interface for Java applications. It allows applications to perform authentication and establish secure communication with the peer. One of the most common security service accessed via the GSS-API and Java GSS-API is Kerberos.

Resources for This Exercise

- Introduction to JAAS and Java GSS-API Tutorials
- Generic Security Service API Version 2: Java Bindings (RFC 2853)
- Java GSS JavaDoc API documentation: `org.ietf.jgss`.

Overview of This Exercise

This exercise is a client-server application that demonstrates how to communicate securely using the Java GSS API. The client and server parts first authenticate to Kerberos, as shown in Exercise 1: Using the JAAS API. This stores the credentials in the subject. The application then executes an action that performs Java GSS operations (with Kerberos as the underlying GSS mechanism) inside a `Subject.callAs` using the subject. The Java GSS Kerberos mechanism, because it is executing inside the `callAs`, obtains the Kerberos credentials from the subject, and uses them to authenticate with the peer and to exchange messages securely.

Steps to Follow

1. Read the `GssServer.java` code.

   This code fragment defines the action to execute after the service principal has authenticated to the KDC. It replaces the `MyAction` in Exercise 1: Using the JAAS API. The code first creates an instance of `GSSManager`, which it uses to obtain its own credentials and to create an instance of `GSSContext`. It uses this context to perform authentication. Upon completing authentication, it accepts encrypted input from the client and uses the established security context to decrypt the data. It then uses the security context to encrypt a reply containing the original input and the date, and then sends it back to the client.
2. Compile the sample code.
3. Read the `GssClient.java` code.

Copyright © 1993, 2026, Oracle and/or its affiliates.

This code fragment defines the action to execute after the client principal has authenticated to the KDC. It replaces the `MyAction` in Exercise 1: Using the JAAS API. The code first creates an instance of **`GSSManager`**, which it uses to obtain a principal name for the service that it is going to communicate with. It then creates an instance of `GSSContext` to perform authentication with the service. Upon completing authentication, it uses the established security context to encrypt a message, and sends it to the server. It then reads an encrypted message from the server and decodes it using the established security context.

4. Compile the sample code.
5. Launch a new window and start the server:

```
% java -Djava.security.auth.login.config=jaas-krb5.conf GssServer
```

6. Run the client application. `GssClient` takes two parameters: the service name and the name of the server that the service is running on. For example, if the service is host running on the machine j1hol-001, you would enter the following:

```
% java -Djava.security.auth.login.config=jaas-krb5.conf GssClient host
j1hol-001
```

When prompted for the password, enter `change_it`.

7. Observe the following output in the respective client and server applications' windows.

Output for running `GssServer` example:

```
Authenticated principal: [host/j1hol-001@J1LABS.EXAMPLE.COM]
Waiting for incoming connections..
Got connection from client /192.0.2.102
Context Established!
Client principal is test@J1LABS.EXAMPLE.COM
Server principal is host/j1hol-001@J1LABS.EXAMPLE.COM
Mutual authentication took place!
Received data "Hello There!" of length 12
Confidentiality applied: true
Sending: Hello There! Thu May 06 12:11:15 PDT 2005
```

Output for running `GssClient` example:

```
Kerberos password for test: change_it
Authenticated principal: [test@J1LABS.EXAMPLE.COM]
Connected to address j1hol-001/192.0.2.102
Context Established!
Client principal is test@J1LABS.EXAMPLE.COM
Server principal is host@j1hol-001
Mutual authentication took place!
Sending message: Hello There!
Will read token of size 93
Received message: Hello There! Thu May 06 12:11:15 PDT 2005
```

Summary

In this exercise, you learned how to write a client-server application that uses the Java GSS API to authenticate and communicate securely with each other.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Next Steps

1. Proceed to Exercise 4: Using the Java SASL API to learn how to write a client/server application that uses the Java SASL API to authenticate and communicate securely with each other.
2. Proceed to Exercise 5: Deploying for Single Sign-On to learn how to configure the sample programs that you have just used to achieve single sign-on in a Kerberos environment.

Exercise 4: Using the Java SASL API

Goal of This Exercise

The goal of this exercise is to learn how to use the Java SASL API to perform secure authentication and communication.

Background for This Exercise

Simple Authentication and Security Layer (SASL) specifies a challenge-response protocol in which data is exchanged between the client and the server for the purposes of authentication and (optional) establishment of a security layer on which to carry on subsequent communications. SASL allows different mechanisms to be used; each such mechanism is identified by a profile that defines the data to be exchanged and a name. SASL is used with connection-based protocols such as LDAPv3 and IMAPv4. SASL is described in RFC 4422.

The Java SASL API defines an API for applications to use SASL in a mechanism-independent way. For example, if you are writing a library for a networking protocol that uses SASL, you can use the Java SASL API to generate the data to be exchanged with the peer. When the library is deployed, you can dynamically configure the mechanisms to use with the library.

In addition to authentication, you can use SASL to negotiate a security layer to be used after authentication. But unlike the GSS-API, the properties of the security layer (such as whether you want integrity or confidentiality) is decided at negotiation time. (the GSS-API allows confidentiality to be turned on or off per message).

Resources for This Exercise

- Java SASL API Programming and Deployment Guide
- `javax.security.sasl`
- Simple Authentication and Security Layer (SASL) (RFC 4422)

Overview of This Exercise

This exercise is a client-server application that demonstrates how to communicate securely using the Java SASL API. The client and server parts first authenticate to Kerberos using Exercise 1: Using the JAAS API. This stores the credentials in the subject. The application then executes an action that performs Java SASL API operations (with Kerberos as the underlying SASL mechanism) inside a `Subject.callAs` using the subject. The SASL/Kerberos mechanism, because it is executing inside the `callAs`, obtains the Kerberos credentials from the subject, and uses them to authenticate with the peer and to exchange messages securely.

This example uses a simple protocol implemented by the `AppConnection` class. This protocol exchanges authentication commands and data commands. Each command consists of a type (e.g., `AppConnection.AUTH_CMD`), the length of the data to follow, and the data itself. The data is

Copyright © 1993, 2026, Oracle and/or its affiliates.

a SASL buffer if it is for authentication or encrypted/integrity-protected application data; it is plain application data otherwise.

Steps to Follow

1. Read the `SaslTestServer.java` sample code.

   This code fragment defines the action to execute after the service principal has authenticated to the KDC. It replaces the `MyAction` in Exercise 1: Using the JAAS API. The server specifies the quality of protections (QOP) that it will support and then creates an instance of `SaslServer` to perform the authentication. The challenge-response protocol of SASL is performed in the while loop, with the server sending challenges to the client and processing the responses from the client. After authentication, the identity of the authenticated client can be obtained via a call to the `getAuthorizedID()` method. If a security layer was negotiated, the server can exchange data securely with the client.

2. Compile the sample code.

3. Read the `SaslTestClient.java` sample code.

   This code fragment defines the action to execute after the client principal has authenticated to the KDC. It replaces the MyAction in Exercise 1: Using the JAAS API. The program first specifies the quality of protections that it wants (in this case, confidentiality) and then creates an instance of `SaslClient` to use for authentication. It then checks whether the mechanism has an initial response and if so, gets the response by invoking the `evaluateChallenge()` method with an empty byte array. It then sends the response to the server to begin the authentication. The challenge-response protocol of SASL is performed in the while loop, with the client evaluating the challenges that it gets from the server and sending the server the corresponding responses to the challenges. After authentication, the client can proceed to communicate with the server using the negotiated security layer.

4. Compile the sample code.

5. Launch a new window and start the server. `SaslTestServer` takes two parameters: the service name and the name of the server that the service is running on. For example, if the service is host running on the machine j1hol-001, you would enter the following:

   ```
   % java -Djava.security.auth.login.config=jaas-krb5.conf SaslTestServer
   host j1hol-001
   ```

6. Run the client application. `SaslTestClient` takes two parameters: the service name and the name of the server that the service is running on. For example, if the service is host running on the machine j1hol-001, you would enter the following:

   ```
   % java -Djava.security.auth.login.config=jaas-krb5.conf SaslTestClient
   host j1hol-001
   ```

   Provide a secure password.

7. Observe the following output in the respective client and server applications' windows.

   Output for running the `SaslTestServer` example:

   ```
   Authenticated principal: [host/j1hol-001@J1LABS.EXAMPLE.COM]
   Waiting for incoming connections...
   Got connection from client /192.0.2.102
   Client authenticated; authorized client is: test@J1LABS.EXAMPLE.COM
   Negotiated QOP: auth-conf
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
Received: Hello There!
Sending: Hello There! Fri May 07 15:32:37 PDT 2005
Received data "Hello There!" of length 12
```

Output for running the `SaslTestClient` example (*password* will be replaced by the password that you provided):

```
Kerberos password for test: password
Authenticated principal: [test@J1LABS.EXAMPLE.COM]
Connected to address j1hol-001/192.0.2.102
Client authenticated.
Negotiated QOP: auth-conf
Sending: Hello There!
Received: Hello There! Fri May 07 15:32:37 PDT 2005
```

Summary

In this exercise, you learned how to write a client-server application that uses the Java SASL API to authenticate and communicate securely with each other.

Next Steps

Proceed to Exercise 5: Deploying for Single Sign-On to learn how to configure the sample programs that you have just used to achieve single sign-on in a Kerberos environment.

Part III : Deploying for Single Sign-On in a Kerberos Environment

Exercise 5: Deploying for Single Sign-On

Goal of This Exercise

The goal of this exercise is to learn how to configure a JAAS application that uses Kerberos for authentication to achieve single sign-on. Single sign-on means that the user needs only authenticate once to a system or a collection of services. After the initial authentication, the user can access other services in the system using the same identity as he used for the initial authentication.

Single sign-on can be used to describe different types of authentication. There are HTTP-based network single sign-on protocols. There is Kerberos-based single sign-on for network services. In this particular exercise, we show how to achieve single sign-on in Kerberos-based systems by showing how to import already-acquired Kerberos credentials from the underlying native operating system.

Background and Resources for This Exercise

See the information provided in Exercise 2: Configuring JAAS for Kerberos Authentication and Exercise 4: Using the Java SASL API for background information about Kerberos and Java GSS.

Steps to Follow

1. Edit the `jaas-krb5.conf` configuration file.

   This file contains two entries: one named *client* and one named *server*. Add the line `useTicketCache=true` to the client entry.

Copyright © 1993, 2026, Oracle and/or its affiliates.

2. Perform Kerberos login to the native operating system. To login to Kerberos, use kinit command as follows:

   ```
   % kinit test
   ```

   Provide a secure password.

3. Run the client and server programs in Exercises 1 through 5 and you will note that the client applications no longer ask you to enter a password.

Part IV : Secure Communications Using Stronger Encryption Algorithms

Exercise 6: Configuring to Use Stronger Encryption Algorithms in a Kerberos Environment, to Secure the Communication

Goal of This Exercise

The goal of this exercise is to learn how to use various Kerberos encryption algorithms to secure the communication. Java GSS/Kerberos provides a wide range of encryption algorithms, including AES256, AES128, 3DES, RC4-HMAC, and DES.

> **Note**
>
> Encryption types based on DES, DES3, and RC4 are disabled by default. See The Kerberos 5 GSS-API Mechanism if you want to use one of these disabled encryption types.

Steps to Follow

1. Configure the Key Distribution Center (KDC) and update the Kerberos database.

   First, you need to update to use the KDC that supports the required Kerberos encryption types.

   You need to update the Kerberos database to generate the new keys with stronger encryption algorithms.

2. If required, edit the Kerberos configuration file (`krb5.conf`).

   You can edit the Kerberos configuration file to select the desired encryption types used. For example, to only enable the AES256-CTS encryption type, add the following:

   ```
   [libdefaults]
   default_tkt_enctypes = aes256-cts
   default_tgs_enctypes = aes256-cts
   permitted_enctypes = aes256-cts
   ```

   To only enable the AES128-CTS encryption type, add the following:

   ```
   [libdefaults]
   default_tkt_enctypes = aes128-cts
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
default_tgs_enctypes = aes128-cts
permitted_enctypes = aes128-cts
```

See The Kerberos 5 GSS-API Mechanism for a list of Java GSS/Kerberos supported encryption types.

> **Note**
>
> Destroy any pre-existing Kerberos TGT in the ticket cache from the previous exercise as follows:
>
> ```
> % kdestroy
> ```

3. Launch a new window and start the server using the updated `krb5.conf` as follows:

```
% java -Djava.security.auth.login.config=jaas-krb5.conf \
-Djava.security.krb5.conf=krb5.conf GSSServer
```

4. Run the client application using the updated `krb5.conf`. The `GSSClient` class takes two parameters: the service name and the name of the server that the service is running on. For example, if the service is host running on the machine j1hol-001, use the following (provide a secure password when prompted):

```
% java -Djava.security.auth.login.config=jaas-krb5.conf \
-Djava.security.krb5.conf=krb5.conf \
GSSClient host j1hol-001
```

Summary

In this exercise, you learned how to write a client-server application that uses Java GSS API to authenticate and communicate securely using stronger Kerberos encryption algorithms. You can enable Kerberos debugging (`-Dsun.security.krb5.debug=true`), to obtain information about the Kerberos encryption type used.

Part V : Secure Authentication Using SPNEGO Java GSS Mechanism

Exercise 7: Using the Java Generic Security Services (GSS) API with SPNEGO

Java GSS is a framework that can support multiple security mechanisms; a way to negotiate a security mechanism underneath GSS-API is needed. This is available via SPNEGO.

SPNEGO is standardized at IETF in RFC 4178. It is a pseudo-security mechanism used to negotiate an underlying security mechanism. It provides the flexibility for client and server to securely negotiate a common GSS security mechanism.

Microsoft makes heavy use of SPNEGO. SPNEGO can be used to inter-operate with Microsoft Server over HTTP, to support HTTP-based cross-platform authentication via the Negotiate Protocol.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Currently, when using Java GSS with Kerberos, we specify the Kerberos OID as follows:

```
Oid krb5Oid = new Oid("1.2.840.113554.1.2.2");
```

In order to use SPNEGO, you only need to specify the SPNEGO OID as follows:

```
Oid spnegoOid = new Oid("1.3.6.1.5.5.2");
```

Then you can use the SPNEGO OID when creating a `GSSCredential`, `GSSContext`, etc.

Goal of This Exercise

Currently the only security mechanism available with Java GSS is Kerberos. The goal of this exercise is to learn how to use other Java GSS mechanisms, such as the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO), to secure the association.

Steps to Follow

1. Read the `GssSpNegoClient.java` code.
2. Compile the sample code:

   ```
   % javac GssSpNegoClient.java
   ```

3. Read the `GssSpNegoServer.java` code.
4. Compile the sample code:

   ```
   % javac GssSpNegoServer.java
   ```

5. Launch a new window and start the server:

   ```
   % java -Djava.security.auth.login.config=jaas-krb5.conf GssSpNegoServer
   ```

6. Run the client application. `GssSpNegoClient` takes two parameters: the service name and the name of the server that the service is running on. For example, if the service is host running on the machine j1hol-001, use the following (provide a secure password when prompted):

   ```
   % java -Djava.security.auth.login.config=jaas-krb5.conf \
   GssSpNegoClient host j1hol-001
   ```

   Sample output for running `GssSpNegoServer`:

   ```
   Authenticated principal: [host/j1hol-001@J1LABS.EXAMPLE.COM]
   Waiting for incoming connections...
   Got connection from client /129.145.128.102
   SPNEGO Negotiated Mechanism = 1.2.840.113554.1.2.2 Kerberos V5
   Context Established!
   Client principal is test@J1LABS.EXAMPLE.COM
   Server principal is
   host/j1hol-001@J1LABS.EXAMPLE.COM
   Mutual authentication took place!
   Received data "Hello There!" of length 12
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
Confidentiality applied: true
Sending: Hello There! Thu May 06 12:11:15 PDT 2005
```

Sample output for running `GssSpNegoClient` (*`password`* is replaced with the password you provided before):

```
Kerberos password for test: password
Authenticated principal: [test@J1LABS.EXAMPLE.COM]
Connected to address j1hol-001/129.145.128.102
SPNEGO Negotiated Mechanism = 1.2.840.113554.1.2.2 Kerberos V5
Context Established!
Client principal is test@J1LABS.EXAMPLE.COM
Server principal is host@j1hol-001
Mutual authentication took place!
Sending message: Hello There!
Will read token of size 93
Received message: Hello There! Thu May 06 12:11:15 PDT 2005
```

Summary

In this exercise, you learned how to write a client-server application that uses the Java GSS API with SPNEGO to negotiate an underlying security mechanism, such as Kerberos, and communicate securely using Kerberos as the underlying authentication system.

> **Note**
>
> Microsoft has implemented certain variations of the SPNEGO protocol. Therefore, to interoperate with Microsoft, a separate mode has been added through the system property `sun.security.spnego.msinterop`. This property is enabled to `true` by default. To disable it, you need to explicitly set this property to `false`. To enable SPNEGO debugging, you can set the system property `sun.security.spnego.debug=true`.

Part VI: HTTP/SPNEGO Authentication

Exercise 8: Using HTTP/SPNEGO Authentication

What is HTTP SPNEGO

HTTP SPNEGO supports the Negotiate authentication scheme in an HTTP communication. SPNEGO supports types of authentication:

Web Authentication

The Web Server responds with

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Negotiate
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

the client will need to send a header like

```
Authorization: Negotiate YY.....
```

to authenticate itself to the server

Proxy Authentication

The Web Server responses with

```
HTTP/1.1 407 Proxy Authentication Required
Proxy-Authenticate: Negotiate
```

the client will need to send a header like

```
Proxy-Authorization: Negotiate YY.....
```

to authenticate itself to the proxy server.

This feature supports both types of authentication.

How to use HTTP/SPNEGO Authentication

There is no new public API function involved in the new feature, but several configurations are needed to perform a success communication:

Kerberos 5 Configuration

Since the SPNEGO mechanism will call JGSS, which in turns calls the Kerberos V5 login module to do real works. Kerberos 5 configurations are needed. This includes the following:

- Some way to provide Kerberos configurations. This can be achieved with the Java system property `java.security.krb5.conf`. For example:

  ```
  java -Djava.security.krb5.conf=krb5.conf \
       -Djavax.security.auth.useSubjectCredsOnly=false \
       ClassName
  ```

  A JAAS config file denoting what login module to use. HTTP SPNEGO codes will look for the standard entry named `com.sun.security.jgss.krb5.initiate`.

  For example, you can provide a file `spnegoLogin.conf`:

  ```
  com.sun.security.jgss.krb5.initiate {
      com.sun.security.auth.module.Krb5LoginModule
          required useTicketCache=true;
  };
  ```

  and run java with:

  ```
  java -Djava.security.krb5.conf=krb5.conf \
       -Djava.security.auth.login.config=spnegoLogin.conf \
  ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        -Djavax.security.auth.useSubjectCredsOnly=false \
        ClassName
```

User Name and Password Retrieval

Just like any other HTTP authentication scheme, the client can provide a customized `java.net.Authenticator` to feed user name and password to the HTTP SPNEGO module **if** they are needed (i.e. there is no credential cache available). The only authentication information needed to be checked in your `Authenticator` is the scheme which can be retrieved with `getRequestingScheme()`. The value should be "Negotiate".

This means your `Authenticator` implementation will look like:

```
class MyAuthenticator extends Authenticator {

        public PasswordAuthentication getPasswordAuthentication () {
            if (getRequestingScheme().equalsIgnoreCase("negotiate")) {
                String krb5user;
                char[] krb5pass;
                // get krb5user and krb5pass in your own way
                ....
                return (new PasswordAuthentication (krb5user,
                            krb5pass));
            } else {
                ....
            }
        }
    }
```

> **Note**
>
> According to the specification of `java.net.Authenticator`, it's designed to get the user name and password at the same time, so do not specify `principal=`*`xxx`* in the JAAS config file.

Scheme Preference

The client can still provide system property `http.auth.preference` to denote that a certain scheme should always be used as long as the server request for it. You can use "SPNEGO" or "Kerberos" for this system property. "SPNEGO" means you prefer to response the Negotiate scheme using the GSS/SPNEGO mechanism; "Kerberos" means you prefer to response the Negotiate scheme using the GSS/Kerberos mechanism. Normally, when authenticating against a Microsoft product, you can use "SPNEGO". The value "Kerberos" also works for Microsoft servers. It's only needed when you encounter a server which knows Negotiate but doesn't know about SPNEGO.

If `http.auth.preference` is not set, the internal order chosen is:

- GSS/SPNEGO -> Digest -> NTLM -> Basic

Notice that Kerberos does not appear in this list, since whenever Negotiate is supported, GSS/SPNEGO is always chosen.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Fallback

If the server has provided more than one authentication scheme (including Negotiate), according to the processing order mentioned in the last section, Java will try to challenge the Negotiate scheme. However, if the protocol cannot be established successfully (for example, the Kerberos configuration is not correct, or the server's hostname is not recorded in the KDC principal DB, or the user name and password provided by `Authenticator` is wrong), then the second strongest scheme will be automatically used.

> **Note**
>
> If `http.auth.preference` is set to SPNEGO or Kerberos, then SPNEGO assumes you only want to try the Negotiate scheme even if it fails. SPNEGO will not fallback to any other scheme and your program will throw an `IOException` saying it received a 401 or 407 error from the HTTP response.

HTTP/SPNEGO Authentication Example

Assume that you have an IIS Server running on a Windows Server within an Active Directory. A web page on this server is configured to be protected by Integrated Windows Authentication. This means the server will prompt for both Negotiate and NTLM authentication.

You need to prepare these files to get the protected file:

RunHttpSpnego.java

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.Authenticator;
import java.net.PasswordAuthentication;
import java.net.URL;

public class RunHttpSpnego {

    static final String kuser = "username"; // your account name
    static final String kpass = "password"; // your password for the account

    static class MyAuthenticator extends Authenticator {
        public PasswordAuthentication getPasswordAuthentication() {
            // I haven't checked getRequestingScheme() here, since for NTLM
            // and Negotiate, the usrname and password are all the same.
            System.err.println("Feeding username and password for " +
getRequestingScheme());
            return (new PasswordAuthentication(kuser, kpass.toCharArray()));
        }
    }

    public static void main(String[] args) throws Exception {
        Authenticator.setDefault(new MyAuthenticator());
        URL url = new URL(args[0]);
        InputStream ins = url.openConnection().getInputStream();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        BufferedReader reader = new BufferedReader(new
InputStreamReader(ins));
        String str;
        while((str = reader.readLine()) != null)
            System.out.println(str);
    }
}
```

krb.conf

```
[libdefaults]
    default_realm = AD.LOCAL
[realms]
    AD.LOCAL = {
        kdc = kdc.ad.local
    }
```

login.conf

```
com.sun.security.jgss.krb5.initiate {
  com.sun.security.auth.module.Krb5LoginModule required  doNotPrompt=false
useTicketCache=true;
};
```

Compiling and Running the Example

1. Compile `RunHttpSpnego.java`.
2. Run `RunHttpSpnego.java`:

   ```
   java -Djava.security.krb5.conf=krb5.conf \
       -Djava.security.auth.login.config=login.conf \
       -Djavax.security.auth.useSubjectCredsOnly=false \
       RunHttpSpnego \
       http://www.ad.local/hello/hello.html
   ```

   You will see the following:

   ```
   Feeding username and password for Negotiate
   <h1>Hello, You got me!</h1>
   ```

   In fact, if you are running on a Windows computer as a domain user, or if you are running on a Linux computer that has already issued the `kinit` command and got the credential cache, then the class `MyAuthenticator` will be completely ignored, and the output will be simply:

   ```
   <h1>Hello, You got me!</h1>
   ```

   which shows the user name and password are not consulted. This is the so-called Single Sign-On.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Also, you can just run

```
java RunHttpSpnego http://www.ad.local/hello/hello.html
```

to see how the fallback is done, in which case you will see

```
Feeding username and password for ntlm
<h1>Hello, You got me!</h1>
```

Source Code for Advanced Security Programming in Java SE Authentication, Secure Communication and Single Sign-On

Jaas.java

```
import javax.security.auth.Subject;
import javax.security.auth.login.*;
import javax.security.auth.callback.CallbackHandler;
import java.security.*;
import com.sun.security.auth.callback.TextCallbackHandler;
import java.io.File;
import java.util.concurrent.Callable;

public class Jaas {
    private static String name;
    private static final boolean verbose = false;

    public static void main(String[] args) throws Exception {
        if (args.length > 0) {
            name = args[0];
        } else {
            name = "client";
        }

        // Create action to perform
        MyAction action = new MyAction();

        loginAndAction(name, action);
    }

    static <T> void loginAndAction(String name, Callable<T> action)
        throws LoginException, PrivilegedActionException {

        // Create a callback handler
        CallbackHandler callbackHandler = new TextCallbackHandler();

        LoginContext context = null;

        try {
            // Create a LoginContext with a callback handler
            context = new LoginContext(name, callbackHandler);

            // Perform authentication
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            context.login();
        } catch (LoginException e) {
            System.err.println("Login failed");
            e.printStackTrace();
            System.exit(-1);
        }

        // Perform action as authenticated user
        Subject subject = context.getSubject();
        if (verbose) {
            System.out.println(subject.toString());
        } else {
            System.out.println("Authenticated principal: " +
                subject.getPrincipals());
        }

        Subject.callAs(subject, action);

        context.logout();
    }

    // Action to perform
    static class MyAction implements Callable<Void> {
        MyAction() { }

        public Void call() throws Exception {
            // Replace the following with an action to be performed
            // by authenticated user
            System.out.println("Performing secure action ...");
            return null;
        }
    }
}
```

jaas-krb5.conf

```
client {
    com.sun.security.auth.module.Krb5LoginModule required
    principal="test";
};

server {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab=sample.keytab
    principal="host/machineName";
};
```

AppConnection.java

```
import java.io.*;
import java.net.Socket;

class AppConnection {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
public static final int AUTH_CMD = 100;
public static final int DATA_CMD = 200;

public static final int SUCCESS = 0;
public static final int AUTH_INPROGRESS = 1;
public static final int FAILURE = 2;

private DataInputStream inStream;
private DataOutputStream outStream;
private Socket socket;

// Client application
AppConnection(String hostName, int port) throws IOException {
    socket = new Socket(hostName, port);

    inStream = new DataInputStream(socket.getInputStream());
    outStream = new DataOutputStream(socket.getOutputStream());

    System.out.println("Connected to address " +
        socket.getInetAddress());
}

// Server side application
AppConnection(Socket socket) throws IOException {
    this.socket = socket;
    inStream = new DataInputStream(socket.getInputStream());
    outStream = new DataOutputStream(socket.getOutputStream());

    System.out.println("Got connection from client " +
        socket.getInetAddress());
}

byte[] receive(int expected) throws IOException {
    if (expected != -1) {
        int cmd = inStream.readInt();
        if (expected != cmd) {
            throw new IOException("Received unexpected code: " + cmd);
        }
        //System.out.println("Read cmd: " + cmd);
    }

    byte[] reply = null;
    int len;
    try {
        len = inStream.readInt();
        //System.out.println("Read length: " + len);

    } catch (IOException e) {
        len = 0;
    }
    if (len > 0) {
        reply = new byte[len];
        inStream.readFully(reply);
    } else {
        reply = new byte[0];
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        return reply;
    }

    AppReply send(int cmd, byte[] bytes) throws IOException {
        //System.out.println("Write cmd: " + cmd);
        outStream.writeInt(cmd);
        if (bytes != null) {
            //System.out.println("Write length: " + bytes.length);
            outStream.writeInt(bytes.length);
            if (bytes.length > 0) {
                outStream.write(bytes);
            }
        } else {
            //System.out.println("Write length: " + 0);
            outStream.writeInt(0);
        }

        outStream.flush();

        if (cmd == SUCCESS || cmd == FAILURE) {
            return null;   // Done
        }

        int returnCode = inStream.readInt();
        //System.out.println("Read cmd: " + returnCode);

        byte[] reply = null;
        if (returnCode != FAILURE) {
            reply = receive(-1);
        }
        return new AppReply(returnCode, reply);
    }

    static class AppReply {
        private int code;
        private byte[] bytes;

        AppReply(int code, byte[] bytes) {
            this.bytes = bytes;
            this.code = code;
        }

        int getStatus() {
            return code;
        }

        byte[] getBytes() {
            return bytes;
        }
    }

    void close() {
        try {
            socket.close();
        } catch (IOException e) {
        }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    }
}
```

GssServer.java

```
import org.ietf.jgss.*;
import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;
import java.security.*;
import java.util.Date;
import java.util.concurrent.Callable;

/**
 * A sample server application that uses JGSS to do mutual authentication
 * with a client using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the client.
 *
 * Every message exchanged with the client includes a 4-byte application-
 * level header that contains the big-endian integer value for the number
 * of bytes that will follow as part of the JGSS token.
 *
 * The protocol is:
 *    1.  Context establishment loop:
 *         a. client sends init sec context token to server
 *         b. server sends accept sec context token to client
 *         ....
 *    2. client sends a wrap token to the server.
 *    3. server sends a wrap token back to the client.
 *
 * Start GssServer first before starting GssClient.
 *
 * Usage:  java <options> GssServer
 *
 * Example: java -Djava.security.auth.login.config=jaas-krb5.conf \
 *               GssServer
 *
 * Add -Djava.security.krb5.conf=krb5.conf to specify application-specific
 * Kerberos configuration (different from operating system's Kerberos
 * configuration).
 */

public class GssServer  {
    private static final int PORT = 4567;
    private static final boolean verbose = false;
    private static final int LOOP_LIMIT = 1;
    private static int loopCount = 0;

    public static void main(String[] args) throws Exception {

        GssServerAction action = new GssServerAction(PORT);

        Jaas.loginAndAction("server", action);
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
static class GssServerAction implements Callable<Void> {
    private int localPort;

    GssServerAction(int port) {
        this.localPort = port;
    }

    public Void call() throws Exception {

        ServerSocket ss = new ServerSocket(localPort);

        // Get own Kerberos credentials for accepting connection
        GSSManager manager = GSSManager.getInstance();
        Oid krb5Mechanism = new Oid("1.2.840.113554.1.2.2");
        GSSCredential serverCreds = manager.createCredential(null,
                                        GSSCredential.DEFAULT_LIFETIME,
                                        krb5Mechanism,
                                        GSSCredential.ACCEPT_ONLY);
        while (loopCount++ < LOOP_LIMIT) {

            System.out.println("Waiting for incoming connection...");

            Socket socket = ss.accept();
            DataInputStream inStream =
                new DataInputStream(socket.getInputStream());

            DataOutputStream outStream =
                new DataOutputStream(socket.getOutputStream());

            System.out.println("Got connection from client " +
                socket.getInetAddress());

            /*
             * Create a GSSContext to receive the incoming request
             * from the client. Use null for the server credentials
             * passed in. This tells the underlying mechanism
             * to use whatever credentials it has available that
             * can be used to accept this connection.
             */

            GSSContext context = manager.createContext(
                (GSSCredential)serverCreds);

            // Do the context establishment loop

            byte[] token = null;

            while (!context.isEstablished()) {

                if (verbose) {
                    System.out.println("Reading ...");
                }
                token = new byte[inStream.readInt()];

                if (verbose) {
                    System.out.println("Will read input token of size " +
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                                token.length + " for processing by
acceptSecContext");
                        }
                        inStream.readFully(token);

                        if (token.length == 0) {
                            if (verbose) {
                                System.out.println("skipping zero length token");
                            }
                            continue;
                        }
                        if (verbose) {
                            System.out.println("Token = " + getHexBytes(token));
                            System.out.println("acceptSecContext..");
                        }
                        token = context.acceptSecContext(token, 0, token.length);

                        // Send a token to the peer if one was generated by
                        // acceptSecContext
                        if (token != null) {
                            if (verbose) {
                                System.out.println("Will send token of size " +
                                    token.length + " from acceptSecContext.");
                            }

                            outStream.writeInt(token.length);
                            outStream.write(token);
                            outStream.flush();
                        }
                    }

                    System.out.println("Context Established! ");
                    System.out.println("Client principal is " +
context.getSrcName());
                    System.out.println("Server principal is " +
context.getTargName());

                    /*
                     * If mutual authentication did not take place, then
                     * only the client was authenticated to the
                     * server. Otherwise, both client and server were
                     * authenticated to each other.
                     */
                    if (context.getMutualAuthState())
                        System.out.println("Mutual authentication took place!");

                    /*
                     * Create a MessageProp which unwrap will use to return
                     * information such as the Quality-of-Protection that was
                     * applied to the wrapped token, whether or not it was
                     * encrypted, etc. Since the initial MessageProp values
                     * are ignored, just set them to the defaults of 0 and false.
                     */
                    MessageProp prop = new MessageProp(0, false);

                    /*
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                     * Read the token. This uses the same token byte array
                     * as that used during context establishment.
                     */
                    token = new byte[inStream.readInt()];
                    if (verbose) {
                        System.out.println("Will read token of size " +
token.length);
                    }
                    inStream.readFully(token);

                    byte[] input = context.unwrap(token, 0, token.length, prop);
                    String str = new String(input, "UTF-8");

                    System.out.println("Received data \"" +
                        str + "\" of length " + str.length());

                    System.out.println("Confidentiality applied: " +
                        prop.getPrivacy());

                    /*
                     * Now generate reply that is the concatenation of the
                     * incoming string with the current time.
                     */

                    /*
                     * First reset the QOP of the MessageProp to 0
                     * to ensure the default Quality-of-Protection
                     * is applied.
                     */
                    prop.setQOP(0);

                    String now = new Date().toString();
                    byte[] nowBytes = now.getBytes("UTF-8");
                    int len = input.length + 1 + nowBytes.length;
                    byte[] reply = new byte[len];
                    System.arraycopy(input, 0, reply, 0, input.length);
                    reply[input.length] = ' ';
                    System.arraycopy(nowBytes, 0, reply, input.length+1,
                        nowBytes.length);

                    System.out.println("Sending: " + new String(reply, "UTF-8"));
                    token = context.wrap(reply, 0, reply.length, prop);

                    outStream.writeInt(token.length);
                    outStream.write(token);
                    outStream.flush();

                    System.out.println("Closing connection with client " +
                        socket.getInetAddress());
                    context.dispose();
                    socket.close();
                }
                return null;
            }
        }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    private static final String getHexBytes(byte[] bytes, int pos, int len) {

        StringBuffer sb = new StringBuffer();
        for (int i = pos; i < (pos+len); i++) {

            int b1 = (bytes[i]>>4) & 0x0f;
            int b2 = bytes[i] & 0x0f;

            sb.append(Integer.toHexString(b1));
            sb.append(Integer.toHexString(b2));
            sb.append(' ');
        }
        return sb.toString();
    }

    private static final String getHexBytes(byte[] bytes) {
        return getHexBytes(bytes, 0, bytes.length);
    }
}
```

GssClient.java

```
import org.ietf.jgss.*;
import java.net.Socket;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.security.*;
import javax.security.auth.login.LoginException;
import java.util.concurrent.Callable;

/**
 * A sample client application that uses JGSS to do mutual authentication
 * with a server using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the server.
 *
 * Every message sent to the server includes a 4-byte application-level
 * header that contains the big-endian integer value for the number
 * of bytes that will follow as part of the JGSS token.
 *
 * The protocol is:
 *    1.  Context establishment loop:
 *         a. client sends init sec context token to server
 *         b. server sends accept sec context token to client
 *         ....
 *    2. client sends a wrapped token to the server.
 *    3. server sends a wrapped token back to the client for the application
 *
 * Start GssServer first before starting GssClient.
 *
 * Usage:  java <options> GssClient <service> <serverName>
 *
 * Example: java -Djava.security.auth.login.config=jaas-krb5.conf \
 *               GssClient host machine.imc.org
 *
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 * Add -Djava.security.krb5.conf=krb5.conf to specify application-specific
 * Kerberos configuration (different from operating system's Kerberos
 * configuration).
 */

public class GssClient {
    private static final int PORT = 4567;
    private static final boolean verbose = false;

    public static void main(String[] args) throws Exception {

        // Obtain the command-line arguments and parse the server's principal

        if (args.length < 2) {
            System.err.println(
                "Usage: java <options> GssClient <service> <serverName>");
            System.exit(-1);
        }

        String serverPrinc = args[0] + "@" + args[1];

        GssClientAction action =
            new GssClientAction(serverPrinc, args[1], PORT);

        Jaas.loginAndAction("client", action);
    }

    static class GssClientAction implements Callable<Void> {
        private String serverPrinc;
        private String hostName;
        private int port;

        GssClientAction(String serverPrinc, String hostName, int port) {
            this.serverPrinc = serverPrinc;
            this.hostName = hostName;
            this.port = port;
        }

        public Void call() throws Exception {
            Socket socket = new Socket(hostName, port);
            DataInputStream inStream =
                new DataInputStream(socket.getInputStream());
            DataOutputStream outStream =
                new DataOutputStream(socket.getOutputStream());

            System.out.println("Connected to address " +
                socket.getInetAddress());

            /*
             * This Oid is used to represent the Kerberos version 5 GSS-API
             * mechanism. It is defined in RFC 1964. We will use this Oid
             * whenever we need to indicate to the GSS-API that it must
             * use Kerberos for some purpose.
             */
            Oid krb5Oid = new Oid("1.2.840.113554.1.2.2");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
GSSManager manager = GSSManager.getInstance();

/*
 * Create a GSSName out of the server's name.
 */
GSSName serverName = manager.createName(serverPrinc,
    GSSName.NT_HOSTBASED_SERVICE);

/*
 * Create a GSSContext for mutual authentication with the
 * server.
 *    - serverName is the GSSName that represents the server.
 *    - krb5Oid is the Oid that represents the mechanism to
 *      use. The client chooses the mechanism to use.
 *    - null is passed in for client credentials
 *    - DEFAULT_LIFETIME lets the mechanism decide how long the
 *      context can remain valid.
 * Note: Passing in null for the credentials asks GSS-API to
 * use the default credentials. This means that the mechanism
 * will look among the credentials stored in the current Subject
 * to find the right kind of credentials that it needs.
 */
GSSContext context = manager.createContext(serverName,
    krb5Oid,
    null,
    GSSContext.DEFAULT_LIFETIME);

// Set the desired optional features on the context. The client
// chooses these options.

context.requestMutualAuth(true);  // Mutual authentication
context.requestConf(true);  // Will use confidentiality later
context.requestInteg(true); // Will use integrity later

// Do the context eastablishment loop

byte[] token = new byte[0];

while (!context.isEstablished()) {

    // token is ignored on the first call
    token = context.initSecContext(token, 0, token.length);

    // Send a token to the server if one was generated by
    // initSecContext
    if (token != null) {
        if (verbose) {
            System.out.println("Will send token of size " +
                token.length + " from initSecContext.");
            System.out.println("writing token = " +
                getHexBytes(token));
        }

        outStream.writeInt(token.length);
        outStream.write(token);
        outStream.flush();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                }

                // If the client is done with context establishment
                // then there will be no more tokens to read in this loop
                if (!context.isEstablished()) {
                    token = new byte[inStream.readInt()];
                    if (verbose) {
                        System.out.println("reading token = " +
                            getHexBytes(token));
                        System.out.println("Will read input token of size " +
                            token.length + " for processing by
initSecContext");
                    }
                    inStream.readFully(token);
                }
            }

            System.out.println("Context Established! ");
            System.out.println("Client principal is " + context.getSrcName());
            System.out.println("Server principal is " +
context.getTargName());

            /*
             * If mutual authentication did not take place, then only the
             * client was authenticated to the server. Otherwise, both
             * client and server were authenticated to each other.
             */
            if (context.getMutualAuthState())
                System.out.println("Mutual authentication took place!");

            byte[] messageBytes = "Hello There!".getBytes("UTF-8");

            /*
             * The first MessageProp argument is 0 to request
             * the default Quality-of-Protection.
             * The second argument is true to request
             * privacy (encryption of the message).
             */
            MessageProp prop =  new MessageProp(0, true);

            /*
             * Encrypt the data and send it across. Integrity protection
             * is always applied, irrespective of confidentiality
             * (i.e., encryption).
             * You can use the same token (byte array) as that used when
             * establishing the context.
             */

            System.out.println("Sending message: " +
                new String(messageBytes, "UTF-8"));
            token = context.wrap(messageBytes, 0, messageBytes.length, prop);
            outStream.writeInt(token.length);
            outStream.write(token);
            outStream.flush();

            /*
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                 * Now we will allow the server to decrypt the message,
                 * append a time/date on it, and send then it back.
                 */

                token = new byte[inStream.readInt()];
                System.out.println("Will read token of size " + token.length);
                inStream.readFully(token);
                byte[] replyBytes = context.unwrap(token, 0, token.length, prop);

                System.out.println("Received message: " +
                    new String(replyBytes, "UTF-8"));

                System.out.println("Done.");
                context.dispose();
                socket.close();

                return null;
            }
        }

        private static final String getHexBytes(byte[] bytes, int pos, int len) {

            StringBuffer sb = new StringBuffer();
            for (int i = pos; i < (pos+len); i++) {

                int b1 = (bytes[i]>>4) & 0x0f;
                int b2 = bytes[i] & 0x0f;

                sb.append(Integer.toHexString(b1));
                sb.append(Integer.toHexString(b2));
                sb.append(' ');
            }
            return sb.toString();
        }

        private static final String getHexBytes(byte[] bytes) {
            return getHexBytes(bytes, 0, bytes.length);
        }
    }
```

SaslTestServer.java

```
import javax.security.sasl.*;
import javax.security.auth.callback.*;
import java.security.*;
import java.util.HashMap;
import java.net.*;
import java.util.Date;
import java.util.concurrent.Callable;

/**
 * A sample server application that uses SASL to authenticate clients
 * using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the client.
 *
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 * This sample program uses a ficticious application-level protocol.
 * Every message exchanged between the client and server an 8-byte
 * header that consists of two integers: the first integer represesents
 * the application-level command or status code while the second integer
 * indicates the length of the SASL buffer. This header is followed by
 * the SASL buffer.
 *
 * The protocol is:
 *    1.  Authentication
 *         a. client sends initial response to server containing
authentication
 *            information
 *         b. server accepts and evaluates response to generate challenge; it
 *            sends the challenge to the server.
 *         c. client evaluates challenge to generate response; it sends the
 *            response;
 *         d. Steps b and c are repeated until authentication succeeds or
fails.
 *    2. client sends an encrypted message to the server.
 *    3. server decryptes the message and sends an encrypted one back
 *       that contains the original message plus the current time.
 *
 * Start SaslTestServer first before starting SaslTestClient.
 *
 * Usage:  java <options> SaslTestServer service serverName
 *
 * Example: java -Djava.security.auth.login.config=jaas-krb5.conf \
 *               SaslTestServer host machine.imc.org
 *
 * Add -Djava.security.krb5.conf=krb5.conf to specify application-specific
 * Kerberos configuration (different from operating system's Kerberos
 * configuration).
 */


public class SaslTestServer {
    private static final String MECH = "GSSAPI"; // SASL name for GSS-API/
Kerberos
    private static final int PORT = 4568;
    private static final int LOOP_LIMIT = 1;
    private static int loopCount = 0;

    public static void main(String[] args) throws Exception {
        // Obtain the command-line arguments and parse the server's principal

        if (args.length < 2) {
            System.err.println(
                "Usage: java <options> SaslTestServer <service> <host>");
            System.exit(-1);
        }

        SaslServerAction action =
            new SaslServerAction(args[0], args[1], PORT);

        Jaas.loginAndAction("server", action);
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    static class SaslServerAction implements Callable<Void> {
        private String service;      // used for SASL authentication
        private String serverName;   // named used for SASL authentication
        private int localPort;
        private CallbackHandler cbh = new TestCallbackHandler();

        SaslServerAction(String service, String serverName, int port) {
            this.service = service;
            this.serverName = serverName;
            this.localPort = port;
        }

        public Void call() throws Exception {
            ServerSocket ss = new ServerSocket(localPort);

            HashMap<String,Object> props = new HashMap<String,Object>();
            props.put(Sasl.QOP, "auth-conf,auth-int,auth");

            // Loop, accepting requests from any client
            while (loopCount++ < LOOP_LIMIT) {
                System.out.println("Waiting for incoming connection...");
                Socket socket = ss.accept();

                // Create application-level connection to handle request
                AppConnection conn = new AppConnection(socket);

                // Normally, the application protocol will negotiate which
                // SASL mechanism to use. In this simplified example, we
                // will always use "GSSAPI", the name of the mechanism that
does
                // Kerberos via GSS-API

                // Create SaslServer to perform authentication
                SaslServer srv = Sasl.createSaslServer(MECH,
                    service, serverName, props, cbh);

                if (srv == null) {
                    throw new Exception(
                        "Unable to find server implementation for " + MECH);
                }

                boolean auth = false;

                // Read initial response from client
                byte[] response = conn.receive(AppConnection.AUTH_CMD);
                AppConnection.AppReply clientMsg;

                while (!srv.isComplete()) {
                    try {
                        // Generate challenge based on response
                        byte[] challenge = srv.evaluateResponse(response);

                        if (srv.isComplete()) {
                            conn.send(AppConnection.SUCCESS, challenge);
                            auth = true;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                        } else {
                            clientMsg =
conn.send(AppConnection.AUTH_INPROGRESS,
                                challenge);
                            response = clientMsg.getBytes();
                        }
                    } catch (SaslException e) {
                        // e.printStackTrace();
                        // Send failure notification to client
                        conn.send(AppConnection.FAILURE, null);
                        break;
                    }
                }

                // Check status of authentication
                if (srv.isComplete() && auth) {
                    System.out.print("Client authenticated; ");
                    System.out.println("authorized client is: " +
                        srv.getAuthorizationID());
                } else {
                    // Go get another client
                    System.out.println("Authentication failed. ");
                    continue;
                }

                String qop = (String) srv.getNegotiatedProperty(Sasl.QOP);
                System.out.println("Negotiated QOP: " + qop);

                // Now try to use security layer
                boolean sl = (qop.equals("auth-conf") || qop.equals("auth-
int"));

                byte[] msg = conn.receive(AppConnection.DATA_CMD);
                byte[] realMsg = (sl ? srv.unwrap(msg, 0, msg.length) : msg);

                System.out.println("Received: " + new String(realMsg,
"UTF-8"));

                // Construct reply to send to client
                String now = new Date().toString();
                byte[] nowBytes = now.getBytes("UTF-8");
                int len = realMsg.length + 1 + nowBytes.length;
                byte[] reply = new byte[len];
                System.arraycopy(realMsg, 0, reply, 0, realMsg.length);
                reply[realMsg.length] = ' ';
                System.arraycopy(nowBytes, 0, reply, realMsg.length+1,
                    nowBytes.length);

                System.out.println("Sending: " + new String(reply, "UTF-8"));

                byte[] realReply = (sl ? srv.wrap(reply, 0, reply.length) :
reply);

                conn.send(AppConnection.SUCCESS, realReply);
            }
            return null;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            }
        }

        static class TestCallbackHandler implements CallbackHandler {

            public void handle(Callback[] callbacks)
                throws UnsupportedCallbackException {

                AuthorizeCallback acb = null;

                for (int i = 0; i < callbacks.length; i++) {
                    if (callbacks[i] instanceof AuthorizeCallback) {
                        acb = (AuthorizeCallback) callbacks[i];
                    } else {
                        throw new UnsupportedCallbackException(callbacks[i]);
                    }
                }

                if (acb != null) {
                    String authid = acb.getAuthenticationID();
                    String authzid = acb.getAuthorizationID();
                    if (authid.equals(authzid)) {
                        // Self is always authorized
                        acb.setAuthorized(true);

                    } else {
                        // Should check some database for mapping and decide.
                        // Current simplified policy is to reject authzids that
                        // don't match authid

                        acb.setAuthorized(false);
                    }

                    if (acb.isAuthorized()) {
                        // Set canonicalized name.
                        // Should look up database for canonical names

                        acb.setAuthorizedID(authzid);
                    }
                }
            }
        }
    }
```

SaslTestClient.java

```
import javax.security.sasl.*;
import javax.security.auth.callback.*;
import java.security.*;
import javax.security.auth.Subject;
import javax.security.auth.login.*;
import com.sun.security.auth.callback.*;
import java.util.HashMap;
import java.util.concurrent.Callable;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
/**
 * A sample client application that uses SASL to authenticate to
 * a server using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the server.
 *
 * This sample program uses a ficticious application-level protocol.
 * Every message exchanged between the client and server an 8-byte
 * header that consists of two integers: the first integer represesents
 * the application-level command or status code while the second integer
 * indicates the length of the SASL buffer. This header is followed by
 * the SASL buffer.
 *
 * The protocol is:
 *    1.  Authentication
 *         a. client sends initial response to server containing
authentication
 *            information
 *         b. server accepts and evaluates response to generate challenge; it
 *            sends the challenge to the server.
 *         c. client evaluates challenge to generate response; it sends the
 *            response;
 *         d. Steps b and c are repeated until authentication succeeds or
fails.
 *    2. client sends an encrypted message to the server.
 *    3. server decryptes the message and sends an encrypted one back
 *       that contains the original message plus the current time.
 *
 * Start SaslTestServer first before starting SaslTestClient.
 *
 * Usage:  java <options> SaslTestClient service serverName
 *
 * Example: java -Djava.security.auth.login.config=jaas-krb5.conf \
 *               SaslTestClient host machine.imc.org
 *
 * Add -Djava.security.krb5.conf=krb5.conf to specify application-specific
 * Kerberos configuration (different from operating system's Kerberos
 * configuration).
 */

public class SaslTestClient {
    private static final String MECH = "GSSAPI"; // SASL name for GSS-API/
Kerberos
    private static final int PORT = 4568;

    private static final byte[] EMPTY = new byte[0];

    public static void main(String[] args) throws Exception {
        // Obtain the command-line arguments and parse the server's principal

        if (args.length < 2) {
            System.err.println(
                "Usage: java <options> SaslTestClient <service>
<serverName>");
            System.exit(-1);
        }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        SaslClientAction action =
            new SaslClientAction(args[0], args[1], PORT);

        Jaas.loginAndAction("client", action);
    }

    static class SaslClientAction implements Callable<Void> {
        private String service;      // used for SASL authentication
        private String serverName;   // name used for SASL authentication
        private int port;
        private CallbackHandler cbh = null; // Don't need handler for GSSAPI

        SaslClientAction(String service, String serverName, int port) {
            this.service = service;
            this.serverName = serverName;
            this.port = port;
        }

        public Void call() throws Exception {
            // Create application-level connection
            AppConnection conn = new AppConnection(serverName, port);

            HashMap<String,Object> props = new HashMap<String,Object>();
            // Request confidentiality
            props.put(Sasl.QOP, "auth-conf");

            // Create SaslClient to perform authentication
            SaslClient clnt = Sasl.createSaslClient(
                new String[]{MECH}, null, service, serverName, props, cbh);

            if (clnt == null) {
                throw new Exception(
                    "Unable to find client implementation for " + MECH);
            }

            byte[] response;
            byte[] challenge;

            // Get initial response for authentication
            response = clnt.hasInitialResponse() ?
                clnt.evaluateChallenge(EMPTY) : EMPTY;

            // Send initial response to server
            AppConnection.AppReply reply =
                conn.send(AppConnection.AUTH_CMD, response);

            // Repeat until authentication terminates
            while (!clnt.isComplete() &&
                (reply.getStatus() == AppConnection.AUTH_INPROGRESS ||
                 reply.getStatus() == AppConnection.SUCCESS)) {

                // Evaluate challenge to generate response
                challenge = reply.getBytes();
                response = clnt.evaluateChallenge(challenge);

                if (reply.getStatus() == AppConnection.SUCCESS) {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                    if (response != null) {
                        throw new Exception("Protocol error interacting with
SASL");
                    }
                    break;
                }

                // Send response to server and read server's next challenge
                reply = conn.send(AppConnection.AUTH_CMD, response);
            }

            // Check status of authentication
            if (clnt.isComplete() && reply.getStatus() ==
AppConnection.SUCCESS) {
                System.out.println("Client authenticated.");
            } else {
                throw new Exception("Authentication failed: " +
                    " connection status? " + reply.getStatus());
            }

            String qop = (String) clnt.getNegotiatedProperty(Sasl.QOP);
            System.out.println("Negotiated QOP: " + qop);

            // Try out security layer
            boolean sl = (qop.equals("auth-conf") || qop.equals("auth-int"));

            byte[] msg = "Hello There!".getBytes("UTF-8");
            System.out.println("Sending: " + new String(msg, "UTF-8"));

            byte[] encrypted = (sl ? clnt.wrap(msg, 0, msg.length) : msg);

            reply = conn.send(AppConnection.DATA_CMD, encrypted);

            if (reply.getStatus() == AppConnection.SUCCESS) {
                byte[] encryptedReply = reply.getBytes();

                byte[] clearReply = (sl ? clnt.unwrap(encryptedReply,
                    0, encryptedReply.length) : encryptedReply);

                System.out.println("Received: " + new String(clearReply,
"UTF-8"));
            } else {
                System.out.println("Failed exchange: " + reply.getStatus());
            }

            conn.close();

            return null;
        }
    }
}
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

krb5.conf

```
# krb5.conf template
# In order to complete this configuration file
# you will need to replace the __<name>__ placeholders
# with appropriate values for your network.
#
[libdefaults]
    default_realm = J1LABS.EXAMPLE.COM
    forwardable = true

[realms]
    J1LABS.EXAMPLE.COM = {
        kdc = j1hol-1280
        kdc = j1hol-004
        admin_server = j1hol-1280
    }

[domain_realm]
    .example.com = J1LABS.EXAMPLE.COM

[logging]
    default = FILE:/var/krb5/kdc.log
    kdc = FILE:/var/krb5/kdc.log
    kdc_rotate = {

# How often to rotate kdc.log. Logs will get rotated no more
# often than the period, and less often if the KDC is not used
# frequently.

        period = 1d

# how many versions of kdc.log to keep around (kdc.log.0, kdc.log.1, ...)

        versions = 10
    }

[appdefaults]
    gkadmin = {
        help_url = http://localhost:8888/ab2/coll.384.1/SEAM
    }
    kinit = {
        renewable = true
        forwardable= true
    }
    rlogin = {
        forwardable= true
    }
    rsh = {
        forwardable= true
    }
    telnet = {
        autologin = true
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        forwardable= true
    }
```

GssSpNegoClient.java

```
import org.ietf.jgss.*;
import java.net.Socket;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.security.*;
import javax.security.auth.login.LoginException;
import java.util.concurrent.Callable;

/**
 * A sample client application that uses JGSS to do mutual authentication
 * with a server using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the server.
 *
 * Every message sent to the server includes a 4-byte application-level
 * header that contains the big-endian integer value for the number
 * of bytes that will follow as part of the JGSS token.
 *
 * The protocol is:
 *    1.  Context establishment loop:
 *         a. client sends init sec context token to server
 *         b. server sends accept sec context token to client
 *         ....
 *    2. client sends a wrapped token to the server.
 *    3. server sends a wrapped token back to the client for the application
 *
 * Start GssServer first before starting GssClient.
 *
 * Usage:  java <options> GssSpNegoClient <service> <serverName>
 *
 * Example: java -Djava.security.auth.login.config=jaas-krb5.conf \
 *               GssSpNegoClient host machine.imc.org
 *
 * Add -Djava.security.krb5.conf=krb5.conf to specify application-specific
 * Kerberos configuration (different from operating system's Kerberos
 * configuration).
 */

public class GssSpNegoClient {
    private static final int PORT = 4567;
    private static final boolean verbose = false;

    public static void main(String[] args) throws Exception {

        // Obtain the command-line arguments and parse the server's principal

        if (args.length < 2) {
            System.err.println(
                "Usage: java <options> GssSpNegoClient <service>
<serverName>");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            System.exit(-1);
        }

        String serverPrinc = args[0] + "@" + args[1];

        GssClientAction action =
            new GssClientAction(serverPrinc, args[1], PORT);

        Jaas.loginAndAction("client", action);
    }

    static class GssClientAction implements Callable<Void> {
        private String serverPrinc;
        private String hostName;
        private int port;

        GssClientAction(String serverPrinc, String hostName, int port) {
            this.serverPrinc = serverPrinc;
            this.hostName = hostName;
            this.port = port;
        }

        public Void call() throws Exception {
            Socket socket = new Socket(hostName, port);
            DataInputStream inStream =
                new DataInputStream(socket.getInputStream());
            DataOutputStream outStream =
                new DataOutputStream(socket.getOutputStream());

            System.out.println("Connected to address " +
                socket.getInetAddress());

            /*
             * This Oid is used to represent the SPNEGO GSS-API
             * mechanism. It is defined in RFC 2478. We will use this Oid
             * whenever we need to indicate to the GSS-API that it must
             * use SPNEGO for some purpose.
             */
            Oid spnegoOid = new Oid("1.3.6.1.5.5.2");

            GSSManager manager = GSSManager.getInstance();

            /*
             * Create a GSSName out of the server's name.
             */
            GSSName serverName = manager.createName(serverPrinc,
                GSSName.NT_HOSTBASED_SERVICE, spnegoOid);

            /*
             * Create a GSSContext for mutual authentication with the
             * server.
             *    - serverName is the GSSName that represents the server.
             *    - krb5Oid is the Oid that represents the mechanism to
             *      use. The client chooses the mechanism to use.
             *    - null is passed in for client credentials
             *    - DEFAULT_LIFETIME lets the mechanism decide how long the
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                 *      context can remain valid.
                 * Note: Passing in null for the credentials asks GSS-API to
                 * use the default credentials. This means that the mechanism
                 * will look among the credentials stored in the current Subject
                 * to find the right kind of credentials that it needs.
                 */
                GSSContext context = manager.createContext(serverName,
                    spnegoOid,
                    null,
                    GSSContext.DEFAULT_LIFETIME);

                // Set the desired optional features on the context. The client
                // chooses these options.

                context.requestMutualAuth(true);  // Mutual authentication
                context.requestConf(true);  // Will use confidentiality later
                context.requestInteg(true); // Will use integrity later

                // Do the context eastablishment loop

                byte[] token = new byte[0];

                while (!context.isEstablished()) {

                    // token is ignored on the first call
                    token = context.initSecContext(token, 0, token.length);

                    // Send a token to the server if one was generated by
                    // initSecContext
                    if (token != null) {
                        if (verbose) {
                            System.out.println("Will send token of size " +
                                token.length + " from initSecContext.");
                            System.out.println("writing token = " +
                                getHexBytes(token));
                        }

                        outStream.writeInt(token.length);
                        outStream.write(token);
                        outStream.flush();
                    }

                    // If the client is done with context establishment
                    // then there will be no more tokens to read in this loop
                    if (!context.isEstablished()) {
                        token = new byte[inStream.readInt()];
                        if (verbose) {
                            System.out.println("reading token = " +
                                getHexBytes(token));
                            System.out.println("Will read input token of size " +
                                token.length + " for processing by
initSecContext");
                        }
                        inStream.readFully(token);
                    }
                }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                System.out.println("Context Established! ");
                System.out.println("Client principal is " + context.getSrcName());
                System.out.println("Server principal is " +
context.getTargName());

                /*
                 * If mutual authentication did not take place, then only the
                 * client was authenticated to the server. Otherwise, both
                 * client and server were authenticated to each other.
                 */
                if (context.getMutualAuthState())
                    System.out.println("Mutual authentication took place!");

                byte[] messageBytes = "Hello There!".getBytes("UTF-8");

                /*
                 * The first MessageProp argument is 0 to request
                 * the default Quality-of-Protection.
                 * The second argument is true to request
                 * privacy (encryption of the message).
                 */
                MessageProp prop =  new MessageProp(0, true);

                /*
                 * Encrypt the data and send it across. Integrity protection
                 * is always applied, irrespective of confidentiality
                 * (i.e., encryption).
                 * You can use the same token (byte array) as that used when
                 * establishing the context.
                 */

                System.out.println("Sending message: " +
                    new String(messageBytes, "UTF-8"));
                token = context.wrap(messageBytes, 0, messageBytes.length, prop);
                outStream.writeInt(token.length);
                outStream.write(token);
                outStream.flush();

                /*
                 * Now we will allow the server to decrypt the message,
                 * append a time/date on it, and send then it back.
                 */

                token = new byte[inStream.readInt()];
                System.out.println("Will read token of size " + token.length);
                inStream.readFully(token);
                byte[] replyBytes = context.unwrap(token, 0, token.length, prop);

                System.out.println("Received message: " +
                    new String(replyBytes, "UTF-8"));

                System.out.println("Done.");
                context.dispose();
                socket.close();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            return null;
        }
    }

    private static final String getHexBytes(byte[] bytes, int pos, int len) {

        StringBuffer sb = new StringBuffer();
        for (int i = pos; i < (pos+len); i++) {

            int b1 = (bytes[i]>>4) & 0x0f;
            int b2 = bytes[i] & 0x0f;

            sb.append(Integer.toHexString(b1));
            sb.append(Integer.toHexString(b2));
            sb.append(' ');
        }
        return sb.toString();
    }

    private static final String getHexBytes(byte[] bytes) {
        return getHexBytes(bytes, 0, bytes.length);
    }
}
```

GssSpNegoServer.java

```
import org.ietf.jgss.*;
import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;
import java.security.*;
import java.util.Date;
import java.util.concurrent.Callable;

/**
 * A sample server application that uses JGSS to do mutual authentication
 * with a client using Kerberos as the underlying mechanism. It then
 * exchanges data securely with the client.
 *
 * Every message exchanged with the client includes a 4-byte application-
 * level header that contains the big-endian integer value for the number
 * of bytes that will follow as part of the JGSS token.
 *
 * The protocol is:
 *    1.  Context establishment loop:
 *         a. client sends init sec context token to server
 *         b. server sends accept sec context token to client
 *         ....
 *    2. client sends a wrap token to the server.
 *    3. server sends a wrap token back to the client.
 *
 * Start GssSpNegoServer first before starting GssClient.
 *
 * Usage:  java <options> GssSpNegoServer
 *
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 * Example: java -Djava.security.auth.login.config=jaas-krb5.conf \
 *               GssSpNegoServer
 *
 * Add -Djava.security.krb5.conf=krb5.conf to specify application-specific
 * Kerberos configuration (different from operating system's Kerberos
 * configuration).
 */

public class GssSpNegoServer  {
    private static final int PORT = 4567;
    private static final boolean verbose = false;
    private static final int LOOP_LIMIT = 1;
    private static int loopCount = 0;

    public static void main(String[] args) throws Exception {

        GssServerAction action = new GssServerAction(PORT);

        Jaas.loginAndAction("server", action);
    }

    static class GssServerAction implements Callable<Void> {
        private int localPort;

        GssServerAction(int port) {
            this.localPort = port;
        }

        public Void call() throws Exception {

            ServerSocket ss = new ServerSocket(localPort);

            // Get own Kerberos credentials for accepting connection
            GSSManager manager = GSSManager.getInstance();
            Oid spnegoOid = new Oid("1.3.6.1.5.5.2");
            GSSCredential serverCreds = manager.createCredential(null,
                                        GSSCredential.DEFAULT_LIFETIME,
                                        spnegoOid,
                                        GSSCredential.ACCEPT_ONLY);
            while (loopCount++ < LOOP_LIMIT) {

                System.out.println("Waiting for incoming connection...");

                Socket socket = ss.accept();
                DataInputStream inStream =
                    new DataInputStream(socket.getInputStream());

                DataOutputStream outStream =
                    new DataOutputStream(socket.getOutputStream());

                System.out.println("Got connection from client " +
                    socket.getInetAddress());

                /*
                 * Create a GSSContext to receive the incoming request
                 * from the client. Use null for the server credentials
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                     * passed in. This tells the underlying mechanism
                     * to use whatever credentials it has available that
                     * can be used to accept this connection.
                     */

                    GSSContext context = manager.createContext(
                        (GSSCredential)serverCreds);

                    // Do the context establishment loop

                    byte[] token = null;

                    while (!context.isEstablished()) {

                        if (verbose) {
                            System.out.println("Reading ...");
                        }
                        token = new byte[inStream.readInt()];

                        if (verbose) {
                            System.out.println("Will read input token of size " +
                                token.length + " for processing by
acceptSecContext");
                        }
                        inStream.readFully(token);

                        if (token.length == 0) {
                            if (verbose) {
                                System.out.println("skipping zero length token");
                            }
                            continue;
                        }
                        if (verbose) {
                            System.out.println("Token = " + getHexBytes(token));
                            System.out.println("acceptSecContext..");
                        }
                        token = context.acceptSecContext(token, 0, token.length);

                        // Send a token to the peer if one was generated by
                        // acceptSecContext
                        if (token != null) {
                            if (verbose) {
                                System.out.println("Will send token of size " +
                                    token.length + " from acceptSecContext.");
                            }

                            outStream.writeInt(token.length);
                            outStream.write(token);
                            outStream.flush();
                        }
                    }

                    System.out.println("Context Established! ");
                    System.out.println("Client principal is " +
context.getSrcName());
                    System.out.println("Server principal is " +
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
context.getTargName());

                /*
                 * If mutual authentication did not take place, then
                 * only the client was authenticated to the
                 * server. Otherwise, both client and server were
                 * authenticated to each other.
                 */
                if (context.getMutualAuthState())
                    System.out.println("Mutual authentication took place!");

                /*
                 * Create a MessageProp which unwrap will use to return
                 * information such as the Quality-of-Protection that was
                 * applied to the wrapped token, whether or not it was
                 * encrypted, etc. Since the initial MessageProp values
                 * are ignored, just set them to the defaults of 0 and false.
                 */
                MessageProp prop = new MessageProp(0, false);

                /*
                 * Read the token. This uses the same token byte array
                 * as that used during context establishment.
                 */
                token = new byte[inStream.readInt()];
                if (verbose) {
                    System.out.println("Will read token of size " +
token.length);
                }
                inStream.readFully(token);

                byte[] input = context.unwrap(token, 0, token.length, prop);
                String str = new String(input, "UTF-8");

                System.out.println("Received data \"" +
                    str + "\" of length " + str.length());

                System.out.println("Confidentiality applied: " +
                    prop.getPrivacy());

                /*
                 * Now generate reply that is the concatenation of the
                 * incoming string with the current time.
                 */

                /*
                 * First reset the QOP of the MessageProp to 0
                 * to ensure the default Quality-of-Protection
                 * is applied.
                 */
                prop.setQOP(0);

                String now = new Date().toString();
                byte[] nowBytes = now.getBytes("UTF-8");
                int len = input.length + 1 + nowBytes.length;
                byte[] reply = new byte[len];
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                    System.arraycopy(input, 0, reply, 0, input.length);
                    reply[input.length] = ' ';
                    System.arraycopy(nowBytes, 0, reply, input.length+1,
                        nowBytes.length);

                    System.out.println("Sending: " + new String(reply, "UTF-8"));
                    token = context.wrap(reply, 0, reply.length, prop);

                    outStream.writeInt(token.length);
                    outStream.write(token);
                    outStream.flush();

                    System.out.println("Closing connection with client " +
                        socket.getInetAddress());
                    context.dispose();
                    socket.close();
                }
                return null;
            }
        }

        private static final String getHexBytes(byte[] bytes, int pos, int len) {

            StringBuffer sb = new StringBuffer();
            for (int i = pos; i < (pos+len); i++) {

                int b1 = (bytes[i]>>4) & 0x0f;
                int b2 = bytes[i] & 0x0f;

                sb.append(Integer.toHexString(b1));
                sb.append(Integer.toHexString(b2));
                sb.append(' ');
            }
            return sb.toString();
        }

        private static final String getHexBytes(byte[] bytes) {
            return getHexBytes(bytes, 0, bytes.length);
        }
    }
```

Appendix A: Setting up Kerberos Accounts

Kerberos accounts are set up on the Key Distribution Center (KDC). Each entry in the Kerberos database contains a Kerberos principal. You should create a host-based principal for the machine that you will be running the servers (for example, "host/j1hol-001") and a client principal (for example, "test") for accessing the servers.

For Windows, see Microsoft Kerberos.

The exercises assume that the operating system has been configured to use the correct Kerberos server. This configuration typically requires administration privileges. If you cannot configure the operating system, then you can use a Kerberos configuration file with your `java`

Copyright © 1993, 2026, Oracle and/or its affiliates.

command by using the `-Djava.security.krb5.conf` option. Here is an example of how to invoke one of the commands from the exercises to use the `krb5.conf` configuration file.

```
% java -Djava.security.auth.login.config=jaas-krb5.conf\
  -Djava.security.krb5.conf=krb5.conf Jaas client
```

The Kerberos 5 GSS-API Mechanism

This section describes and lists security features regarding Java Generic Security Services (Java GSS) for Kerberos 5. It also describes the Object Identifier (OID) for the Kerberos V5 mechanism, the encryption types, and the `krb5.conf` settings supported by Java GSS.

The Generic Security Services Application Program Interface (GSS-API) mechanism is defined by RFC 1964 and supplemented with RFC 4121 under the Internet Standards process.

The OID for the Kerberos V5 Mechanism

According to RFC 1964 section 1, the OID for Java Generic Security Services (Java GSS) for Kerberos 5 is defined as 1.2.840.113554.1.2.2; see also GSSAPI Mechanisms in *Java Security Standard Algorithm Names*.

Java GSS/Kerberos Supported Encryption Types

The following table lists the preferred order of Java GSS/Kerberos supported encryption types.

Table 7-1 Java GSS/Kerberos Supported Encryption Types

Name	Aliases	etype Number
aes256-cts-hmac-sha1-96	aes256-sha1, aes256-cts	18
aes128-cts-hmac-sha1-96	aes128-sha1, aes128-cts	17
aes256-cts-hmac-sha384-192	aes256-sha2	20
aes128-cts-hmac-sha256-128	aes128-sha2	19
des3-cbc-sha1	des3-hmac-sha1	16
arcfour-hmac-md5	arcfour-hmac, rc4-hmac	23
des-cbc-crc	*None*	1
des-cbc-md5	*None*	3

Note

The AES-128 and AES-256 encryption types are enabled by default. The following legacy encryption types are disabled by default:

- DES-based encryption types, including des-cbc-crc and dec-cbc-md5
- The DES3-based encryption type, des3-cbc-sha1
- The RC4-based encryption type, arcfour-hmac-md5 (alias rc4-hmac)

If you want to use one of these disabled encryption types, then in your Kerberos configuration file, you must set the parameter `allow_weak_crypto` to `true` and specify its encryption type in the parameters `default_tkt_enctypes`, `default_tgs_enctypes`, and `permitted_enctypes`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Supported krb5.conf Settings

The following parameters are supported. A user can restrict the usage of encryption for various purposes in `krb5.conf`, in the `[libdefaults]` section.

```
include FILENAME
includedir DIRNAME

[libdefaults]
allow_weak_crypto
canonicalize
clockskew
default_keytab_name
default_realm
default_tgs_enctypes
default_tkt_enctypes
dns_canonicalize_hostname
dns_fallback
dns_lookup_kdc
dns_lookup_realm
extra_addresses
forwardable
kdc_default_options
kdc_timeout
max_retries
no_addresses
noaddresses
permitted_enctypes
proxiable
renew_lifetime
renewable
ticket_lifetime
udp_preference_limit

[realms]
  REALM.NAME = {
      kdc
      kdc_timeout
      udp_preference_limit
      max_retries
  }

[capaths]
  A = {
    I = .
    B = I
  }

[domain_realm]
  domain=REALM
```

The following are the default values for `krb5.conf` file parameters:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-2 Default Values for krb5.conf file parameters

Parameter	Default Value
`allow_weak_crypto`	`false`
`canonicalize`	`false`
`clockskew`	`300`
`default_tgs_enctypes`	*Value of permitted encryption types*
`default_tkt_enctypes`	*Value of permitted encryption types*
`dns_canonicalize_hostname`	`true`
`dns_lookup_kdc`	`true`
`dns_lookup_realm`	`false`
`forwardable`	`false`
`kdc_timeout`	`30s`
`max_retries`	`3`
`no_addresses`	`true`
`noaddresses`	`true`
`permitted_enctypes`	*Encryption types enabled by default, AES-128 and AES-256; see* Table 7-1>
`proxiable`	`false`
`renewable`	`false`
`udp_preference_limit`	`1465`

If no `krb5.conf` file is found or a setting doesn't exist in a `krb5.conf` file, then these default values will be used. For example, a DNS lookup will be performed to fetch KDC details because the default value of `dns_lookup_kdc` is `true`.

Kerberos System Properties, Security Properties, and Environment Variables

The following table lists system properties, security properties, and environment variables related to Kerberos.

> **Note**
>
> Some properties are both a security property and a system property. If you set a value for the system property, then it overrides any value set as a security property. See How to Specify a java.lang.System Property and How to Specify a java.security.Security Property.

Table 7-3 Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
`KRB5CCNAME`	Environment variable	Specifies the complete path name to the Kerberos credentials cache (CC) file.	The default value is operating system-dependent. For Linux and macOS, it is `/tmp/krb5cc_`*`userid`*

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
LOGONSERVER	Environment variable	On Windows, the default KDC if a `krb5.conf` file cannot be found or it does not contain settings for the default realm's KDC. See Locating the krb5.conf Configuration File	See Locating the krb5.conf Configuration File
USERDNSDOMAIN	Environment variable	On Windows, the default realm if a `krb5.conf` file cannot be found or it does not contain settings for the default realm. See Locating the krb5.conf Configuration File.	See Locating the krb5.conf Configuration File

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
`jdk.security.krb5.default.initiate.credential`	Security property and system property	Specifies the policy for the proxy_impersonator Kerberos ccache configuration entry. The proxy_impersonator ccache configuration entry indicates that the ccache is a synthetic delegated credential for use with the Service for User to Proxy (S4U2proxy) Kerberos extension by an intermediate server. The ccache file should also contain the Ticket Granting Ticket (TGT) of this server and an evidence ticket from the default principal of the ccache to this server. The value of this security property determines how Java uses this configuration entry. It can have one of the following values: • `no-impersonate`: Ignore this configuration entry, and always act as the owner of the TGT (if it exists). • `try-impersonate`: Try impersonation when this configuration entry exists. If no matching TGT or evidence ticket is found, fallback to no-impersonate. • `always-impersonate`: Always impersonate when this configuration entry exists. If no matching TGT or evidence ticket is found, no initial credential is read from the ccache.	`always-impersonate`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
`jdk.security.krb5.name.case.sensitive`	Security property and system property	If `true`, then the comparison of principal names during keytab and credentials cache (ccache) entry lookup is case-sensitive. The default value is `false` to ensure backward compatibility. When looking up a keytab or ccache entry for a Kerberos principal, the principal name is compared with the entry name in a case-insensitive manner. However, many Kerberos implementations treat principal names as case-sensitive. As a result, if two principals have names that differ only by case, there is a risk of selecting the incorrect keytab or ccache entry.	`false`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
`jdk.security.krb5.s4u2proxy.acceptNonForwardableServiceTicket`	Security property and system property	Specifies the policy for a nonforwardable service ticket in a S4U2proxy request. The Service for User to Proxy (S4U2proxy) Kerberos extension enables a middle service to obtain a service ticket to another service on behalf of a user. It requires that the user's service ticket to the first service has the forwardable flag set. However, some KDC implementations ignore this requirement and accept service tickets with the flag unset. If this security property is set to `true`, then: • The user service ticket, when obtained by the middle service after a S4U2self impersonation, is not required to have the forwardable flag. • If a S4U2proxy request receives a KRB_ERROR of the KDC_ERR_BADOPTION error code and the ticket to the middle service is not forwardable, then the JDK will try the same request with another KDC instead of treating it as a fatal failure.	`false`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
`krb5.kdc.bad.policy`	Security property and system property	Specifies the policy for failed Kerberos KDC lookups. When a KDC is unavailable (because of, for example, a network error or service failure), it is put inside a secondary list and accessed less often for future requests. The value of this security property can be one of the following: • `tryLast`: KDCs in the secondary list are always tried after those not on the list. Example: `krb5.kdc.bad.policy = tryLast` • `tryLess[:max_retries,timeout]`: KDCs in the secondary list are still tried by their order in the configuration, but with smaller *`max_retries`* and *`timeout`* values. *`max_retries`* and *`timeout`* are optional numerical parameters (default 1 and 5000, which means once and 5 seconds). Note that if any of the values defined here are more than what is defined in `krb5.conf`, they will be ignored. Example: `krb5.kdc.bad.policy = tryLess:2,2000` Whenever a KDC is detected as available, it is removed from the secondary list. The secondary list is reset when `krb5.conf` is reloaded. You can add `refreshKrb5Config=true` to a JAAS	`tryLast`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
		configuration file so that `krb5.conf` is reloaded whenever a JAAS authentication is attempted.	
`sun.security.krb5.disableReferrals`	Security property and system property	Disable Kerberos cross-realm referrals. The JDK's Kerberos client supports cross-realm referrals as defined in RFC 6806: Kerberos Principal Name Canonicalization and Cross-Realm Referrals. This allows to setup more dynamic environments in which clients do not need to know in advance how to reach the realm of a target principal (either a user or service). When a client issues a Kerberos Authentication Service (AS) or a Ticket-Granting Service (TGS) request, the "canonicalize" option is set to announce support of this feature. A KDC server may fulfill the request or reply referring the client to a different one. If referred, the client will issue a new request and the cycle repeats. In addition to referrals, the "canonicalize" option allows the KDC server to change the client name in response to an AS request. For security reasons the FAST scheme is enforced. See RFC 6806, section 11, "Negotiation of FAST and Detecting Modified Requests."	`false`
`sun.security.krb5.maxReferrals`	Security property and system property	Maximum number of AS or TGS referrals to avoid infinite loops.	5

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 7-3 (Cont.) Kerberos Properties and Environment Variables

Name	Type	Description	Default Value
`java.security.krb5.conf`	System property	Specifies the location of the `krb5.conf` file. See Locating the krb5.conf Configuration File for more information.	None
`java.security.krb5.kdc`	System property	Specifies the default KDC for the default realm specified by `java.security.krb5.realm`. See Setting Properties to Indicate the Default Realm and KDC for more information.	None
`java.security.krb5.realm`	System property	Specifies the default realm. See Setting Properties to Indicate the Default Realm and KDC for more information.	None

Troubleshooting JGSS

See Enabling Debugging in Java Generic Security Services for more information about how to enable debugging in JGSS, Kerberos, SPNEGO, the native JGSS bridge, and the SSPI bridge on Windows.

Troubleshooting Logins

The following are some problems that may occur when attempting a login, and suggestions for solving them.

- **Configurable Kerberos Settings:** The Kerberos Key Distribution Center (KDC) name and realm settings are provided in the Kerberos configuration file or via the system properties `java.security.krb5.kdc` and `java.security.krb5.realm`. A boolean option `refreshKrb5Config` can be specified in the entry for `Krb5LoginModule` in the JAAS configuration file. If this option is set to `true`, then the configuration values will be refreshed before the login method of the `Krb5LoginModule` is called.

  > **Note**
  >
  > When switching Kerberos configurations, it is REQUIRED that `refreshKrb5Config` should be set to `true`. Failure to set this value can lead to unexpected results.

- **java.lang.SecurityException at javax.security.auth.login.Configuration.getConfiguration**

  Cause: There was a problem processing the JAAS login configuration file, possibly due to a syntax error in the file.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Solution: Check the configuration file carefully for errors. See Appendix B: JAAS Login Configuration File for information about the syntax required in the login configuration file.

- **javax.security.auth.login.LoginException: KrbException: Pre-authentication information was invalid (24) - Preauthentication failed**

  Cause 1: The password entered is incorrect.

  Solution 1: Verify the password.

  Cause 2: If you are using the keytab to get the key (e.g., by setting the `useKeyTab` option to `true` in the Krb5LoginModule entry in the JAAS login configuration file), then the key might have changed since you updated the keytab.

  Solution 2: Consult your Kerberos documentation to generate a new keytab and use that keytab.

  Cause 3: Clock skew - If the time on the KDC and on the client differ significantly (typically 5 minutes), this error can be returned.

  Solution 3: Synchronize the clocks (or have a system administrator do so).*Cause 4:* The Kerberos realm name is not all uppercase.

  Solution 4: Make the Kerberos realm name all uppercase. **Note**: It is recommended to have all uppercase realm names. See Naming Conventions for Realm Names and Hostnames.

- **GSSException: No valid credentials provided (Mechanism level: Attempt to obtain new INITIATE credentials failed! (null)) . . . Caused by: javax.security.auth.login.LoginException: Clock skew too great**

  Cause: Kerberos requires the time on the KDC and on the client to be loosely synchronized. (The default is within 5 minutes.) If that's not the case, you will get this error.

  Solution: Synchronize the clocks (or have a system administrator do so).

- **javax.security.auth.login.LoginException: KrbException: Null realm name (601) - default realm not specified**

  Cause: The default realm is not specified in the Kerberos configuration file `krb5.conf` (if used), provided as a part of the user name, or specified via the `java.security.krb5.realm` system property.

  Solution: Verify that your Kerberos configuration file (if used) contains an entry specifying the default realm, or directly specify it by setting the value of the `java.security.krb5.realm` system property and/or including it in your user name when authenticating using Kerberos.

- **javax.security.auth.login.LoginException: java.net.SocketTimeoutException: Receive timed out**

  Solution: Verify that the Kerberos KDC is up and running.

- **GSSException: No valid credentials provided (Mechanism level: Failed to find any Kerberos Ticket)**

  Cause: This may occur if no valid Kerberos credentials are obtained. In particular, this occurs if you want the underlying mechanism to obtain credentials but you forgot to indicate this by setting the `javax.security.auth.useSubjectCredsOnly` system property value to `false` (for example via `-Djavax.security.auth.useSubjectCredsOnly=false` in your execution command).

  Solution: Be sure to set the `javax.security.auth.useSubjectCredsOnly` system property value to `false` if you want the underlying mechanism to obtain credentials, rather than your

Copyright © 1993, 2026, Oracle and/or its affiliates.

application or a wrapper program (such as the Login utility used by some of the tutorials) performing authentication using JAAS.

- **javax.security.auth.login.LoginException: Could not load configuration file <krb5.conf> (No such file or directory)**

  Cause: The tutorials' sample execution commands specify the default Kerberos realm and KDC by setting values for the `java.security.krb5.realm` and `java.security.krb5.kdc` system properties. If you like, you can instead have a `krb5.conf` Kerberos configuration file used. Such a file includes information about what the default realm and KDC are. To use a `krb5.conf` file, you either set the system property `java.security.krb5.conf` (instead of the `realm` and `kdc` properties) to specify the location of the file or you don't set any of these properties and therefore an attempt is made to locate the `krb5.conf` file in a default location. You will get the error "Could not load configuration file <krb5.conf> (No such file or directory)" if the file could not be found.

  Solution: Verify that the Kerberos configuration file `krb5.conf` is available and readable. Check Kerberos Requirements for information about how to specify the location of the `krb5.conf` file and where such a file is searched for by default if you don't explicitly indicate the location.

- **javax.security.auth.login.LoginException: KrbException: KDC has no support for encryption type (14) - KDC has no support for encryption type**

  Cause 1: Your KDC does not support the encryption type requested.

  Solution 1: Oracle's implementation of Kerberos supports the following encryption types: `aes256-cts-hmac-sha1-96`, `aes128-cts-hmac-sha1-96`, `des3-cbc-sha1`, `arcfour-hmac-md5`, `des-cbc-crc`, and `des-cbc-md5`.

  Applications can select the desired encryption type by specifying following tags in the Kerberos Configuration file `krb5.conf`:

  ```
  [libdefaults]
  default_tkt_enctypes = des-cbc-md5 des-cbc-crc des3-cbc-sha1
  default_tgs_enctypes = des-cbc-md5 des-cbc-crc des3-cbc-sha1
  permitted_enctypes = des-cbc-md5 des-cbc-crc des3-cbc-sha1
  ```

  If not specified, then the default value is:

  ```
  aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96 des3-cbc-sha1 arcfour-hmac-md5
  ```

  If `allow_weak_crypto` in `krb5.conf` is set to true, then `des-cbc-crc` and `des-cbc-md5` are also supported.

  Cause 2: This exception is thrown when using native ticket cache on some Windows platforms. Microsoft has added a new feature in which they no longer export the session keys for Ticket-Granting Tickets (TGTs). As a result, the native TGT obtained on Windows has an "empty" session key and null EType.

  Solution 2: You need to update the Windows registry to disable this new feature. The registry key **`allowtgtsessionkey`** should be added – and set correctly – to allow session keys to be sent in the Kerberos Ticket-Granting Ticket. Usually, the following is the required registry setting:

  ```
  HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\Kerberos\Parameters
  Value Name: allowtgtsessionkey
  ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
Value Type: REG_DWORD
Value: 0x01  ( default is 0 )
```

By default, the value is 0; setting it to "0x01" allows a session key to be included in the TGT.

- **KDC reply did not match expectations**

  Cause: The KDC sent a response that cannot be understood by the client.

  Solution: Verify that you have set correctly all the `krb5.conf` file configuration parameters and consult your KDC vendor's guide.

> **Note**
>
> A debugging mode can be enabled by setting the system property `sun.security.krb5.debug` to "true". This setting allows you to follow the program's execution of the Kerberos V5 protocol.

Copyright © 1993, 2026, Oracle and/or its affiliates.

8

Java Secure Socket Extension (JSSE) Reference Guide

The Java Secure Socket Extension (JSSE) enables secure Internet communications. It provides a framework and an implementation for a Java version of the TLS and DTLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication.

Introduction to JSSE

Data that travels across a network can easily be accessed by someone who is not the intended recipient. When the data includes private information, such as passwords and credit card numbers, steps must be taken to make the data unintelligible to unauthorized parties. It is also important to ensure that the data has not been modified, either intentionally or unintentionally, during transport. The Transport Layer Security (TLS) protocol was designed to help protect the privacy and integrity of data while it is being transferred across a network.

The Java Secure Socket Extension (JSSE) enables secure Internet communications. It provides a framework and an implementation for a Java version of the TLS protocol and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Using JSSE, developers can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, or FTP) over TCP/IP.

By abstracting the complex underlying security algorithms and handshaking mechanisms, JSSE minimizes the risk of creating subtle but dangerous security vulnerabilities. Furthermore, it simplifies application development by serving as a building block that developers can integrate directly into their applications.

JSSE provides both an application programming interface (API) framework and an implementation of that API. The JSSE API supplements the core network and cryptographic services defined by the `java.security` and `java.net` packages by providing extended networking socket classes, trust managers, key managers, SSL contexts, and a socket factory framework for encapsulating socket creation behavior. Because the `SSLSocket` class is based on a blocking I/O model, the Java Development Kit (JDK) includes a nonblocking `SSLEngine` class to enable implementations to choose their own I/O methods.

The JSSE API supports the following security protocols:

- DTLS: versions 1.0 and 1.2
- TLS: version 1.0, 1.1, 1.2, and 1.3
- SSL (Secure Socket Layer): version 3.0

These security protocols encapsulate a normal bidirectional stream socket, and the JSSE API adds transparent support for authentication, encryption, and integrity protection.

JSSE is a security component of the Java SE platform, and is based on the same design principles found elsewhere in the Java Cryptography Architecture (JCA) Reference Guide framework. This framework for cryptography-related security components allows them to have

Copyright © 1993, 2026, Oracle and/or its affiliates.

implementation independence and, whenever possible, algorithm independence. JSSE uses the Cryptographic Service Providers defined by the JCA framework.

Other security components in the Java SE platform include the Java Authentication and Authorization Service (JAAS) Reference Guide and the Java security tools (see Security Tools Summary). JSSE encompasses many of the same concepts and algorithms as those in JCA but automatically applies them underneath a simple stream socket API.

The JSSE API was designed to allow other SSL/TLS/DTLS protocols and Public Key Infrastructure (PKI) implementations to be plugged in seamlessly. Developers can also provide alternative logic to determine if remote hosts should be trusted or what authentication key material should be sent to a remote host.

JSSE Features and Benefits

JSSE includes the following important benefits and features:

- Included as a standard component of the JDK
- Extensible, provider-based architecture
- Implemented in 100% pure Java
- Provides API support for TLS/DTLS
- Provides implementations of SSL 3.0, TLS (versions 1.0, 1.1, 1.2, and 1.3), and DTLS (versions 1.0 and 1.2)
- Includes classes that can be instantiated to create secure channels (`SSLSocket`, `SSLServerSocket`, and `SSLEngine`)
- Provides support for cipher suite negotiation, which is part of the TLS/DTLS handshaking used to initiate or verify secure communications
- Provides support for client and server authentication, which is part of the normal TLS/DTLS handshaking
- Provides support for HTTP encapsulated in the TLS protocol, which allows access to data such as web pages using HTTPS
- Provides server session management APIs to manage memory-resident SSL sessions
- Provides support for the certificate status request extension (OCSP stapling), which saves client certificate validation round-trips and resources
- Provides support for the Server Name Indication (SNI) extension, which extends the TLS/DTLS protocols to indicate what server name the client is attempting to connect to during handshaking
- Provides support for endpoint identification during handshaking, which prevents man-in-the-middle attacks
- Provides support for cryptographic algorithm constraints, which provides fine-grained control over algorithms negotiated by JSSE

JSSE Standard API

The JSSE standard API, available in the `javax.net` and `javax.net.ssl` packages, provides:

- Secure sockets tailored to client and server-side applications.
- A non-blocking engine for producing and consuming streams of TLS/DTLS data (`SSLEngine`).

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Factories for creating sockets, server sockets, SSL sockets, and SSL server sockets. By using socket factories, you can encapsulate socket creation and configuration behavior.
- A class representing a secure socket context that acts as a factory for secure socket factories and engines.
- Key and trust manager interfaces (including X.509-specific key and trust managers), and factories that can be used for creating them.
- A class for secure HTTP URL connections (HTTPS).

SunJSSE Provider

Oracle's implementation of Java SE includes a JSSE provider named *SunJSSE*, which comes preinstalled and preregistered with the JCA. This provider supplies the following cryptographic services:

- An implementation of the SSL 3.0, TLS (versions 1.0, 1.1, 1.2, and 1.3), and DTLS (versions 1.0 and 1.2) security protocols.
- An implementation of the most common TLS and DTLS cipher suites. This implementation encompasses a combination of authentication, key agreement, encryption, and integrity protection.
- An implementation of an X.509-based key manager that chooses appropriate authentication keys from a standard JCA keystore.
- An implementation of an X.509-based trust manager that implements rules for certificate chain validation.

See The SunJSSE Provider.

JSSE Related Documentation

The following list contains links to online documentation and names of books about related subjects:

JSSE API Documentation

- `javax.net` package
- `javax.net.ssl` package

Java SE Security

- The Java SE Security home page
- The Security Features in Java SE trail of the Java Tutorial
- Java PKI Programmer's Guide
- Inside Java 2 Platform Security, Second Edition: Architecture, API Design and Implementation

Transport Layer Security (TLS)

- The TLS Protocol Version 1.0
- The TLS Protocol Version 1.1
- The TLS Protocol Version 1.2
- The (TLS) Protocol Version 1.3

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Transport Layer Security (TLS) Extensions
- HTTP Over TLS

Datagram Transport Layer Security (DTLS)

- The DTLS Protocol Version 1.0
- The DTLS Protocol Version 1.2

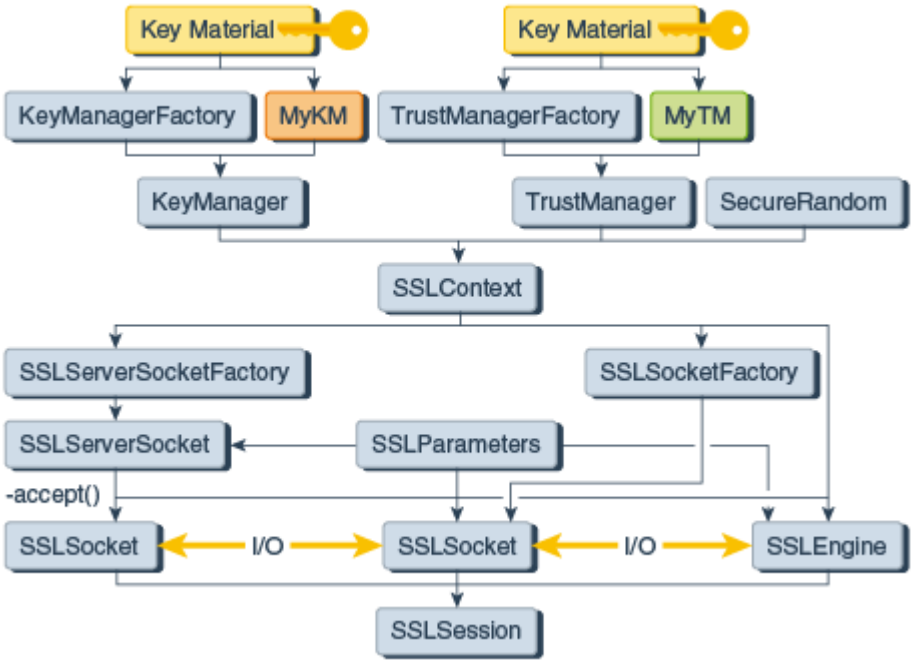
U.S. Encryption Policies

- U.S. Department of Commerce
- Technology CEO Council
- Current export policies: Encryption and Export Administration Regulations (EAR)
- NIST Computer Security Publications

JSSE Classes and Interfaces

To communicate securely, both sides of the connection must be SSL-enabled. In the JSSE API, the endpoint classes of the connection are `SSLSocket` and `SSLEngine`. In Figure 8-1, the major classes used to create `SSLSocket` and `SSLEngine` are laid out in a logical ordering.

Figure 8-1 JSSE Classes Used to Create SSLSocket and SSLEngine



An `SSLSocket` is created either by an `SSLSocketFactory` or by an `SSLServerSocket` accepting an inbound connection. In turn, an `SSLServerSocket` is created by an `SSLServerSocketFactory`. Both `SSLSocketFactory` and `SSLServerSocketFactory` objects are created by an `SSLContext`. An `SSLEngine` is created directly by an `SSLContext`, and relies on the application to handle all I/O.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> When using raw `SSLSocket` or `SSLEngine` classes, you should always check the peer's credentials before sending any data. Since JDK 7, endpoint identification/verification procedures can be handled during SSL/TLS handshaking. See the method `SSLParameters.setEndpointIdentificationAlgorithm`.
> For example, the host name should match the host name in the peer's credentials. An application could be exploited with host name spoofing if the host name is not verified.

JSSE Core Classes and Interfaces

The core JSSE classes are part of the `javax.net` and `javax.net.ssl` packages.

SocketFactory and ServerSocketFactory Classes

The abstract `javax.net.SocketFactory` class is used to create sockets. Subclasses of this class are factories that create particular subclasses of sockets and thus provide a general framework for the addition of public socket-level functionality. For example, see SSLSocketFactory and SSLServerSocketFactory Classes.

The abstract `javax.net.ServerSocketFactory` class is analogous to the `SocketFactory` class, but is used specifically for creating server sockets.

Socket factories are a simple way to capture a variety of policies related to the sockets being constructed, producing such sockets in a way that does not require special configuration of the code that asks for the sockets:

- Due to polymorphism of both factories and sockets, different kinds of sockets can be used by the same application code just by passing different kinds of factories.
- Factories can themselves be customized with parameters used in socket construction. For example, factories could be customized to return sockets with different networking timeouts or security parameters already configured.
- The sockets returned to the application can be subclasses of `java.net.Socket` (or `javax.net.ssl.SSLSocket`), so that they can directly expose new APIs for features such as compression, security, record marking, statistics collection, or firewall tunneling.

SSLSocketFactory and SSLServerSocketFactory Classes

The `javax.net.ssl.SSLSocketFactory` class acts as a factory for creating secure sockets. This class is an abstract subclass of `javax.net.SocketFactory`.

Secure socket factories encapsulate the details of creating and initially configuring secure sockets. This includes authentication keys, peer certificate validation, enabled cipher suites, and the like.

The `javax.net.ssl.SSLServerSocketFactory` class is analogous to the `SSLSocketFactory` class, but is used specifically for creating server sockets.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Obtaining an SSLSocketFactory

The following ways can be used to obtain an `SSLSocketFactory`:

- Get the default factory by calling the `SSLSocketFactory.getDefault()` static method.
- Receive a factory as an API parameter. That is, code that must create sockets but does not care about the details of how the sockets are configured can include a method with an `SSLSocketFactory` parameter that can be called by clients to specify which `SSLSocketFactory` to use when creating sockets (for example, `javax.net.ssl.HttpsURLConnection`).
- Construct a new factory with specifically configured behavior.

The default factory is typically configured to support server authentication only so that sockets created by the default factory do not leak any more information about the client than a normal TCP socket would.

Many classes that create and use sockets do not need to know the details of socket creation behavior. Creating sockets through a socket factory passed in as a parameter is a good way of isolating the details of socket configuration, and increases the reusability of classes that create and use sockets.

You can create new socket factory instances either by implementing your own socket factory subclass or by using another class which acts as a factory for socket factories. One example of such a class is `SSLContext`, which is provided with the JSSE implementation as a provider-based configuration class.

SSLSocket and SSLServerSocket Classes

The `javax.net.ssl.SSLSocket` class is a subclass of the standard Java `java.net.Socket` class. It supports all of the standard socket methods and adds methods specific to secure sockets. Instances of this class encapsulate the `SSLContext` under which they were created. See SSLContext Class. There are APIs to control the creation of secure socket sessions for a socket instance, but trust and key management are not directly exposed.

The `javax.net.ssl.SSLServerSocket` class is analogous to the `SSLSocket` class, but is used specifically for creating server sockets.

To prevent peer spoofing, you should always verify the credentials presented to an `SSLSocket`. See Cipher Suite Choice and Remote Entity Verification.

> **Note**
>
> Due to the complexity of the SSL and TLS protocols, it is difficult to predict whether incoming bytes on a connection are handshake or application data, and how that data might affect the current connection state (even causing the process to block). In the Oracle JSSE implementation, the `available()` method on the object obtained by `SSLSocket.getInputStream()` returns a count of the number of application data bytes successfully decrypted from the SSL connection but not yet read by the application.

Obtaining an SSLSocket

Instances of `SSLSocket` can be obtained in one of the following ways:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- An `SSLSocket` can be created by an instance of `SSLSocketFactory` via one of the several `createSocket` methods of that class.
- An `SSLSocket` can be created through the `accept` method of the `SSLServerSocket` class.

Cipher Suite Choice and Remote Entity Verification

The SSL/TLS protocols define a specific series of steps to ensure a *protected* connection. However, the choice of cipher suite directly affects the type of security that the connection enjoys. For example, if an anonymous cipher suite is selected, then the application has no way to verify the remote peer's identity. If a suite with no encryption is selected, then the privacy of the data cannot be protected. Additionally, the SSL/TLS protocols do not specify that the credentials received must match those that peer might be expected to send. If the connection were somehow redirected to a rogue peer, but the rogue's credentials were acceptable based on the current trust material, then the connection would be considered valid.

When using raw `SSLSocket` and `SSLEngine` classes, you should always check the peer's credentials before sending any data. The `SSLSocket` and `SSLEngine` classes do not automatically verify that the host name matches the host name in the peer's credentials. An application could be exploited with host name spoofing if the host name is not verified. Since JDK 7, endpoint identification/verification procedures can be handled during SSL/TLS handshaking. See the `SSLParameters.getEndpointIdentificationAlgorithm` method.

Protocols such as HTTPS (HTTP Over TLS) do require host name verification. Since JDK 7, the HTTPS endpoint identification is enforced during handshaking for `HttpsURLConnection` by default. See the `SSLParameters.getEndpointIdentificationAlgorithm` method. Alternatively, applications can use the `HostnameVerifier` interface to override the default HTTPS host name rules. See HostnameVerifier Interface and HttpsURLConnection Class.

SSLEngine Class

As mentioned previously, TLS/DTLS are standard protocols for secure network communications, and are being used in a wide variety of applications across a wide range of computing platforms and devices. Along with this popularity come demands to use TLS/DTLS with different I/O and threading models to satisfy the applications' performance, scalability, footprint, and other requirements. There are demands to use TLS/DTLS with blocking and nonblocking I/O channels, asynchronous I/O, arbitrary input and output streams, and byte buffers. There are demands to use it in highly scalable, performance-critical environments, requiring management of thousands of network connections.

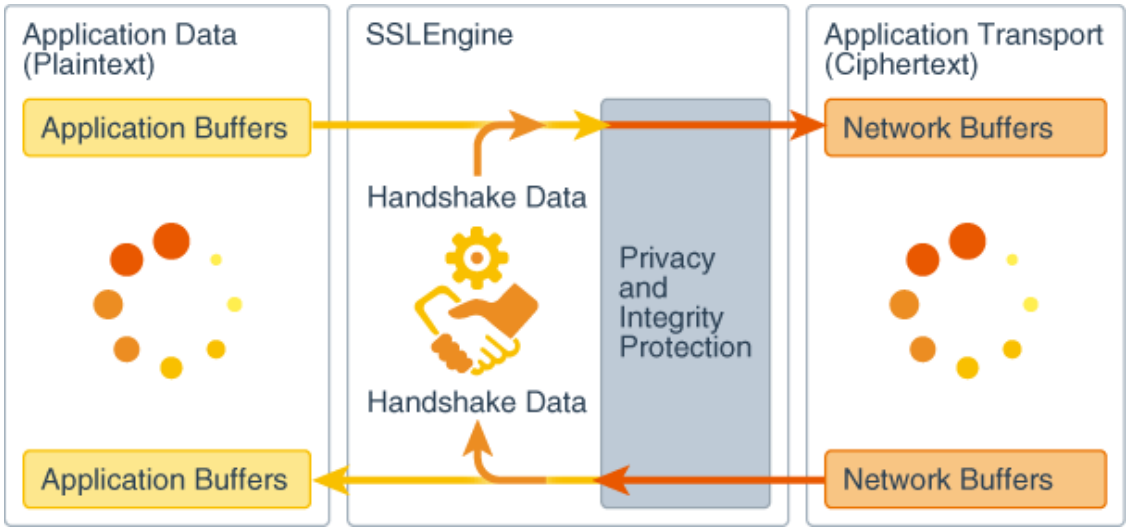
Abstracting the I/O transport mechanism using the `SSLEngine` class in Java SE allows applications to use the TLS/DTLS protocols in a transport-independent way, and thus frees application developers to choose transport and computing models that best meet their needs. Not only does this abstraction allow applications to use nonblocking I/O channels and other I/O models, it also accommodates different threading models. This effectively leaves the I/O and threading decisions up to the application developer. Because of this flexibility, the application developer must manage I/O and threading (complex topics in and of themselves), as well as have some understanding of the TLS/DTLS protocols. Because the `SSLEngine` class requires an understanding of SSL/TLS, I/O, and threading models, it is considered an advanced API: beginners should use `SSLSocket`.

Users of other Java programming language APIs such as the Java Generic Security Services (Java GSS-API) and the Java Simple Authentication Security Layer (Java SASL) will notice similarities in that the application is also responsible for transporting data.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The core class is `javax.net.ssl.SSLEngine`. It encapsulates a TLS/DTLS state machine and operates on inbound and outbound byte buffers that are filled and drained respectively by the user of the `SSLEngine` class. Figure 8-2 illustrates the flow of data from the application, through `SSLEngine`, to the transport mechanism, and back.

Figure 8-2 Flow of Data Through SSLEngine



Calls to `SSLEngine` produce and consume TLS/DTLS packets, which must then be exchanged with the peer. The nomenclature for `SSLEngine` data is always from the perspective of the local side: data bound for the peer is called *outbound data*, and the peer's data for the local side is called *inbound data*. Before application data can be produced/consumed from the application buffers, a handshaking procedure that negotiates security parameters must complete. The handshake data that is produced/consumed is internal to `SSLEngine` and must be exchanged with the peer before application data will be produced/consumed. The application is responsible for all data transportation.

The application, shown on the left, supplies application (plaintext) data in an application buffer and passes it to `SSLEngine`. After handshaking has completed and cryptographic parameters have been negotiated, the `SSLEngine` object consumes the data contained in the outbound application data buffer to produce TLS/DTLS encoded data and places it in the network buffer supplied by the application. The application is now responsible for sending the contents of the outbound network buffer to the peer using the transport mechanism. Upon receiving TLS/DTLS encoded data from its peer (via the transport), the application places the inbound data into a network buffer and passes it to `SSLEngine`. The `SSLEngine` object processes the network buffer's contents to produce application data (or handshake data, which is consumed internally).

An instance of the `SSLEngine` class can be in one of the following states:

Creation
The `SSLEngine` has been created and initialized, but has not yet been used. During this phase, an application may set any `SSLEngine`-specific settings (enabled cipher suites, whether the `SSLEngine` should handshake in client or server mode, and so on). Once handshaking has begun, though, any new settings (except client/server mode) will be used for the next handshake.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Initial handshaking
The initial handshake is a procedure by which the two peers exchange communication parameters until an `SSLSession` is established. Application data can't be sent during this phase.

Application data
After the communication parameters have been established and the handshake is complete, application data can flow through the `SSLEngine`. Outbound application messages are encrypted and integrity protected, and inbound messages reverse the process.

Rehandshaking
Either side can request a renegotiation of the session at any time during the Application Data phase. New handshaking data can be intermixed among the application data. Before starting the rehandshake phase, the application may reset the TLS/DTLS communication parameters such as the list of enabled cipher suites and whether to use client authentication, but can not change between client/server modes. As before, after handshaking has begun, any new `SSLEngine` configuration settings won't be used until the next handshake.

Closure
When the connection is no longer needed, the application should close the `SSLEngine` and should send/receive any remaining messages to the peer before closing the underlying transport mechanism. Once an engine is closed, it is not reusable: a new `SSLEngine` must be created.

SSLEngine Methods

There are three types of `SSLEngine` methods: those that initialize the `SSLEngine` and start the handshake, those that process data packets for writing to or reading from the network, and those that properly close the `SSLEngine` and connection.

The following steps describe the handshake process with respect to the methods of `SSLEngine`:

1. After you have created the `SSLEngine`, call the various `set*` methods to configure all aspects of the connection that is about to occur (for example, `setEnabledProtocols()`, `setEnabledCipherSuites()`, `setUseClientMode()`, and `setWantClientAuth()`). You can also configure the connection with the `SSLParameters` class, which enables you to set multiple settings in a single method call.
2. Obtain the currently empty `SSLSession` for the `SSLEngine`, then determine the maximum buffer sizes for the application and network bytes that could be generated with the `getApplicationBufferSize()` and `getPacketBufferSize()` methods. Allocate `ByteBuffer` instances for application and network buffers accordingly.
3. Once you have configured the connection and the buffers, call the `beginHandshake()` method, which moves the `SSLEngine` into the initial handshaking state.
4. Create the transport mechanism that the connection will use with, for example, the `SocketChannel` or `Socket` classes.
5. Call the `wrap()` and `unwrap()` methods to perform the initial handshaking. You'll need to call these methods several times before application data can be consumed, produced, and properly protected by later `wrap()`/`unwrap()` calls.

   The handshake bytes must be exchanged with the peer using the transport mechanism. For more information about the TLS handshaking mechanism, see one of the TLS RFCs (such as RFC 5246: The Transport Layer Security (TLS) Protocol: Version 1.2).

Copyright © 1993, 2026, Oracle and/or its affiliates.

For example, if your SSLEngine is acting as a client and handshaking using TLSv1.2, then you might see the following occurring:

a. The wrap() method produces a TLS ClientHello message, then places it in the outbound network buffer. The application must correctly send the bytes of this message to the peer.

b. The SSLEngine must now process the peer's response (such as the ServerHello, Certificate, and ServerHelloDone messages) to drive the handshake forward. The application obtains the response bytes from the network transport and places them in the inbound network buffer. The SSLEngine processes these bytes using the unwrap() method.

c. The SSLEngine sends more handshaking data (such as the ChangeCipherSuite and Finished messages). The wrap() places the bytes of the message in the outbound network buffer. The application must correctly send these bytes to the peer as before.

d. The SSLEngine waits for its peer's ChangeCipherSuite or Finished message. The bytes of this message follow the same path as in Step b.

6. Once the handshaking has completed, application data can now start flowing. Call the wrap() method to take the bytes from the outbound application buffer, encrypt and protect them, and then place them in the network buffer for transport to the peer. Likewise, call the unwrap() method to decrypt and unprotect inbound network data. The resulting application data is placed in the inbound application data buffer.

7. Once data has been exchanged between the two peers, close both the inbound and outbound sides of the SSLEngine. Call the closeOutbound() method to signal the SSLEngine that the application will not be sending any more data. Call the closeInbound() method to signal the SSLEngine that the network connection has been closed and there will be no more data.

Understanding SSLEngine Operation Statuses

To indicate the status of the engine and what actions the application should take, the SSLEngine.wrap() and SSLEngine.unwrap() methods return an SSLEngineResult instance, as shown in Example 8-5. This SSLEngineResult object contains two pieces of status information: the overall status of the engine and the handshaking status.

The possible overall statuses are represented by the SSLEngineResult.Status enum. The following statuses are available:

OK
There was no error.

CLOSED
The operation closed the SSLEngine or the operation could not be completed because it was already closed.

BUFFER_UNDERFLOW
The input buffer had insufficient data to process, indicating that the application must obtain more data from the peer (for example, by reading more data from the network) and try the operation again.

BUFFER_OVERFLOW
The output buffer had insufficient space to hold the result, indicating that the application must clear or enlarge the destination buffer and try the operation again.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 8-1 illustrates how to handle the `BUFFER_UNDERFLOW` and `BUFFER_OVERFLOW` statuses of the `SSLEngine.unwrap()` method. It uses `SSLSession.getApplicationBufferSize()` and `SSLSession.getPacketBufferSize()` to determine how large to make the byte buffers.

The possible handshaking statuses are represented by the `SSLEngineResult.HandshakeStatus` enum. They represent whether handshaking has completed, whether the caller must send or receive more handshaking data from the peer, and so on. The following handshake statuses are available:

`FINISHED`
The `SSLEngine` has just finished handshaking.

`NEED_TASK`
The `SSLEngine` needs the results of one (or more) delegated tasks before handshaking can continue.

`NEED_UNWRAP`
The `SSLEngine` needs to receive data from the remote side before handshaking can continue.

`NEED_UNWRAP_AGAIN`
The `SSLEngine` needs to unwrap before handshaking can continue. This value indicates that not-yet-interpreted data has been previously received from the remote side and does not need to be received again; the data has been brought into the JSSE framework but has not been processed yet.

`NEED_WRAP`
The `SSLEngine` must send data to the remote side before handshaking can continue, so `SSLEngine.wrap()` should be called.

`NOT_HANDSHAKING`
The `SSLEngine` is not currently handshaking.

Having two statuses per result allows the `SSLEngine` to indicate that the application must take two actions: one in response to the handshaking and one representing the overall status of the `wrap()` and `unwrap()` methods. For example, the engine might, as the result of a single `SSLEngine.unwrap()` call, return `SSLEngineResult.Status.OK` to indicate that the input data was processed successfully and `SSLEngineResult.HandshakeStatus.NEED_UNWRAP` to indicate that the application should obtain more TLS/DTLS encoded data from the peer and supply it to `SSLEngine.unwrap()` again so that handshaking can continue. As you will see, the following examples are greatly simplified; they would need to be expanded significantly to properly handle all of these status combinations.

Example 8-2 and Example 8-3 illustrate how to process handshaking data by checking handshaking status and the overall status of the `wrap()` and `unwrap()` methods.

Example 8-1 Sample Code for Handling BUFFER_UNDERFLOW and BUFFER_OVERFLOW

The following code sample illustrates how to handle BUFFER_UNDERFLOW and BUFFER_OVERFLOW status:

```
SSLEngineResult res = engine.unwrap(peerNetData, peerAppData);
switch (res.getStatus()) {

    case BUFFER_OVERFLOW:
        // Maybe need to enlarge the peer application data buffer if
        // it is too small, and be sure you've compacted/cleared the
        // buffer from any previous operations.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            if (engine.getSession().getApplicationBufferSize() >
peerAppData.capacity()) {
                // enlarge the peer application data buffer
            } else {
                // compact or clear the buffer
            }
            // retry the operation
        break;

        case BUFFER_UNDERFLOW:
            // Not enough inbound data to process. Obtain more network data
            // and retry the operation. You may need to enlarge the peer
            // network packet buffer, and be sure you've compacted/cleared
            // the buffer from any previous operations.
            if (engine.getSession().getPacketBufferSize() >
peerNetData.capacity()) {
                // enlarge the peer network packet buffer
            } else {
                // compact or clear the buffer
            }
            // obtain more inbound network data and then retry the operation
            break;

        // Handle other status: CLOSED, OK
        // ...
    }
```

Example 8-2 Sample Code for Checking and Processing Handshaking Statuses and Overall Statuses

The following code sample illustrates how to process handshaking data by checking handshaking status and the overall status of the `wrap()` and `unwrap()` methods:

```
void doHandshake(SocketChannel socketChannel, SSLEngine engine,
    ByteBuffer myNetData, ByteBuffer peerNetData) throws Exception {

    // Create byte buffers to use for holding application data
    int appBufferSize = engine.getSession().getApplicationBufferSize();
    ByteBuffer myAppData = ByteBuffer.allocate(appBufferSize);
    ByteBuffer peerAppData = ByteBuffer.allocate(appBufferSize);

    // Begin handshake
    engine.beginHandshake();
    SSLEngineResult.HandshakeStatus hs = engine.getHandshakeStatus();

    // Process handshaking message
    while (hs != SSLEngineResult.HandshakeStatus.FINISHED &&
        hs != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {

        switch (hs) {

        case NEED_UNWRAP:
            // Receive handshaking data from peer
            if (socketChannel.read(peerNetData) < 0) {
                // The channel has reached end-of-stream
            }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                // Process incoming handshaking data
                peerNetData.flip();
                SSLEngineResult res = engine.unwrap(peerNetData, peerAppData);
                peerNetData.compact();
                hs = res.getHandshakeStatus();

                // Check status
                switch (res.getStatus()) {
                case OK :
                    // Handle OK status
                    break;

                   // Handle other status: BUFFER_UNDERFLOW, BUFFER_OVERFLOW,
CLOSED
                   // ...
                }
                break;

            case NEED_WRAP:
                // Ensure that any previous net data in myNetData has been sent
                // to the peer (not shown here), then generate more.

                // Empty/clear the local network packet buffer.
                myNetData.clear();

                // Generate more data to send if possible.
                res = engine.wrap(myAppData, myNetData);
                hs = res.getHandshakeStatus();

                // Check status
                switch (res.getStatus()) {
                case OK :
                    myNetData.flip();

                    // Send the handshaking data to peer
                    while (myNetData.hasRemaining()) {
                        socketChannel.write(myNetData);
                    }
                    break;

                    // Handle other status:  BUFFER_OVERFLOW, BUFFER_UNDERFLOW,
CLOSED
                    // ...
                }
                break;

            case NEED_TASK :
                // Handle blocking tasks
                break;

                // Handle other status:  // FINISHED or NOT_HANDSHAKING
                // ...
            }
        }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    // Processes after handshaking
    // ...
}
```

Example 8-3 Sample Code for Handling DTLS handshake Status and Overall Status

The following code sample illustrates how to handle DTLS handshake status:

```
void handshake(SSLEngine engine, DatagramSocket socket,
               SocketAddress peerAddr) throws Exception {
    boolean endLoops = false;
    // private static int MAX_HANDSHAKE_LOOPS = 60;
    int loops = MAX_HANDSHAKE_LOOPS;
    engine.beginHandshake();
    while (!endLoops && (serverException == null) && (clientException ==
null)) {
        if (--loops < 0) {
            throw new RuntimeException("Too many loops to produce handshake
packets");
        }
        SSLEngineResult.HandshakeStatus hs = engine.getHandshakeStatus();
        if (hs == SSLEngineResult.HandshakeStatus.NEED_UNWRAP ||
                hs == SSLEngineResult.HandshakeStatus.NEED_UNWRAP_AGAIN) {
            ByteBuffer iNet;
            ByteBuffer iApp;
            if (hs == SSLEngineResult.HandshakeStatus.NEED_UNWRAP) {
                // receive ClientHello request and other SSL/TLS/DTLS records
                byte[] buf = new byte[1024];
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                try {
                    socket.receive(packet);
                } catch (SocketTimeoutException ste) {
                    // retransmit the packet if timeout
                    List <DatagramPacket> packets =
                        onReceiveTimeout(engine, peerAddr);
                    for (DatagramPacket p : packets) {
                        socket.send(p);
                    }
                    continue;
                }
                iNet = ByteBuffer.wrap(buf, 0, packet.getLength());
                iApp = ByteBuffer.allocate(1024);
            } else {
                iNet = ByteBuffer.allocate(0);
                iApp = ByteBuffer.allocate(1024);
            }
            SSLEngineResult r = engine.unwrap(iNet, iApp);
            SSLEngineResult.Status rs = r.getStatus();
            hs = r.getHandshakeStatus();
            if (rs == SSLEngineResult.Status.BUFFER_OVERFLOW) {
                // the client maximum fragment size config does not work?
                throw new Exception("Buffer overflow: " +
                                    "incorrect client maximum fragment size");
            } else if (rs == SSLEngineResult.Status.BUFFER_UNDERFLOW) {
                // bad packet, or the client maximum fragment size
                // config does not work?
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                if (hs != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {
                    throw new Exception("Buffer underflow: " +
                                        "incorrect client maximum fragment
size");
                } // otherwise, ignore this packet
            } else if (rs == SSLEngineResult.Status.CLOSED) {
                endLoops = true;
            }   // otherwise, SSLEngineResult.Status.OK:
            if (rs != SSLEngineResult.Status.OK) {
                continue;
            }
        } else if (hs == SSLEngineResult.HandshakeStatus.NEED_WRAP) {
            List <DatagramPacket> packets =
                // Call a function to produce handshake packets
                produceHandshakePackets(engine, peerAddr);
            for (DatagramPacket p : packets) {
                socket.send(p);
            }
        } else if (hs == SSLEngineResult.HandshakeStatus.NEED_TASK) {
            runDelegatedTasks(engine);
        } else if (hs == SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {
            // OK, time to do application data exchange.
            endLoops = true;
        } else if (hs == SSLEngineResult.HandshakeStatus.FINISHED) {
            endLoops = true;
        }
    }
    SSLEngineResult.HandshakeStatus hs = engine.getHandshakeStatus();
    if (hs != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {
        throw new Exception("Not ready for application data yet");
    }
}
```

SSLEngine for TLS Protocols

This section shows you how to create an `SSLEngine` object and use it to generate and process TLS data.

Creating an SSLEngine Object

Use the `SSLContext.createSSLEngine()` method to create an `SSLEngine` object.

Before you use an `SSLEngine` object, you must configure the engine to act as a client or a server, and set other configuration parameters, such as which cipher suites to use and whether client authentication is required.

Example 8-4 Sample Code for Creating an SSLEngine Client for TLS with PKCS12 as Keystore

The following sample code creates an `SSLEngine` client for TLS that uses PKCS12 as keystore.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> In this sample, the server name and port number are not used for communicating with the server (all transport is the responsibility of the application). They are hints to the JSSE provider to use for TLS session caching.

```
import javax.net.ssl.*;
import java.security.*;

// Create and initialize the SSLContext with key material
char[] passphrase = "passphrase".toCharArray();

// First initialize the key and trust material
KeyStore ksKeys = KeyStore.getInstance("PKCS12");
ksKeys.load(new FileInputStream("testKeys"), passphrase);
KeyStore ksTrust = KeyStore.getInstance("PKCS12");
ksTrust.load(new FileInputStream("testTrust"), passphrase);

// KeyManagers decide which key material to use
KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
kmf.init(ksKeys, passphrase);

// TrustManagers decide whether to allow connections
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(ksTrust);

// Get an instance of SSLContext for TLS protocols
sslContext = SSLContext.getInstance("TLS");
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

// Create the engine
SSLEngine engine = sslContext.createSSLengine(hostname, port);

// Use as client
engine.setUseClientMode(true);
```

Generating and Processing TLS Data

The two main `SSLEngine` methods are `wrap()` and `unwrap()`. They are responsible for generating and consuming network data respectively. Depending on the state of the `SSLEngine` object, this data might be handshake or application data.

Performing TLS Handshake, Then Processing TLS Data, With SSLEngine.wrap() and SSLEngine.unwrap()

Each `SSLEngine` object has several phases during its lifetime. Before application data can be sent or received, the TLS protocol requires a handshake to establish cryptographic parameters. This handshake requires a series of back-and-forth steps by the `SSLEngine` object. During the initial handshaking, the `wrap()` and `unwrap()` methods generate and consume handshake data before starting to exchange application data.

The application is responsible for reliably transporting the data (for example, by using TCP) to and from the peer. That is, your application (and not `SSLEngine`) must reliably deliver to the

Copyright © 1993, 2026, Oracle and/or its affiliates.

peer any data generated by the `wrap()` method, and your application (and not `SSLEngine`) must reliably obtain data from the peer so that it can decode it by calling the `unwrap()` method.

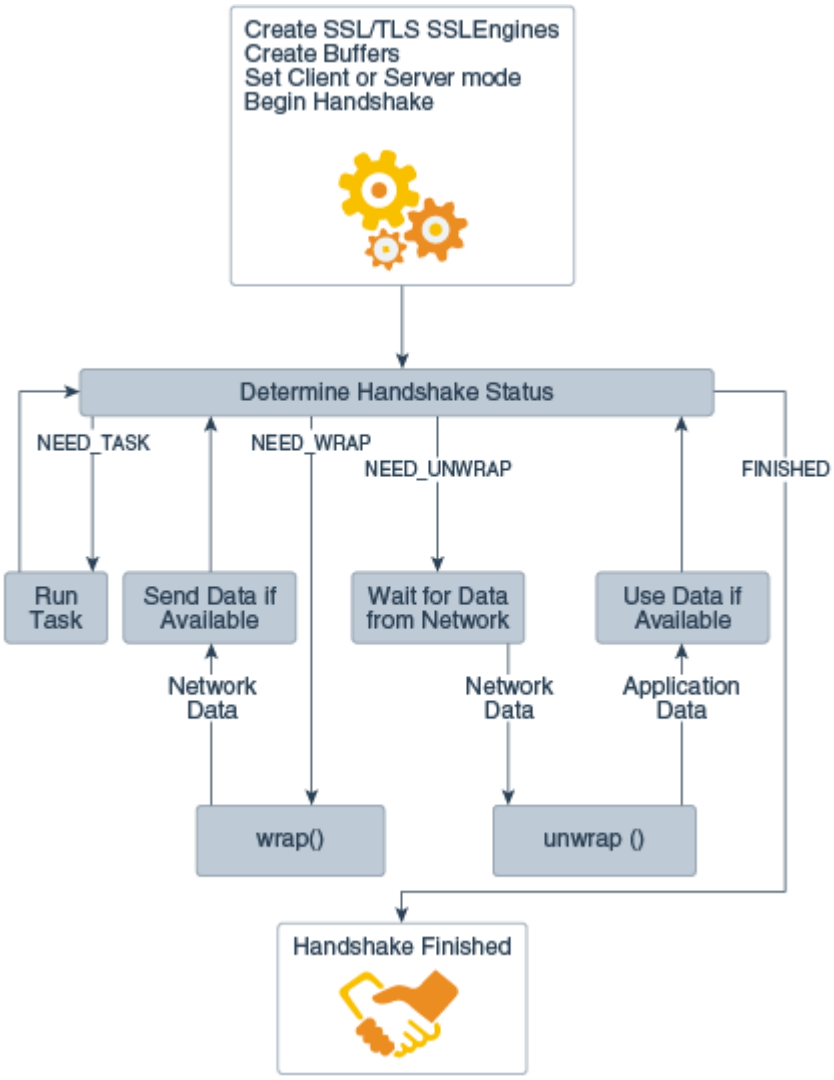
Each `SSLEngine` operation generates an instance of the `SSLEngineResult` class, in which the `SSLEngineResult.HandshakeStatus` field is used to determine what operation must occur next to move the handshake along.

When handshaking is complete, further calls to `wrap()` will attempt to consume application data and package it for transport. The `unwrap()` method will attempt the opposite.

To send data to the peer, the application first supplies the data that it wants to send via `SSLEngine.wrap()` to obtain the corresponding TLS encoded data. The application then sends the encoded data to the peer using its chosen transport mechanism. When the application receives the TLS encoded data from the peer via the transport mechanism, it supplies this data to the `SSLEngine` via `SSLEngine.unwrap()` to obtain the plaintext data sent by the peer.

Figure 8-3 shows the state machine during a typical TLS handshake, with corresponding messages and statuses:

Figure 8-3 State Machine during TLS Handshake



Copyright © 1993, 2026, Oracle and/or its affiliates.

Steps to Process TLS Data

Suppose you want to process TLS data sent between a client and server. In general, you would follow these steps:

1. Create `ByteBuffer` instances to represent the application data buffer and the network data buffer on the client and server. In the client's and server's outbound application data buffers, specify the data that you want encrypted and sent over the network to the server and client, respectively.

   > **Note**
   >
   > In the `wrap(ByteBuffer src, ByteBuffer dst)` method, the parameter `src` is the application data buffer and `dst` is the network data buffer. Conversely, in the `unwrap(ByteBuffer src, ByteBuffer dst)` method, the parameter `src` is the network data buffer and `dst` is the application data buffer. Both `wrap()` and `unwrap()` return an instance of `SSLEngineResult`, which contains a `SSLEngineResult.HandshakeStatus` field that indicates whether the handshake is complete or what must occur next to move the handshake along.

2. In a loop, call `wrap()` and `unwrap()` on the client and server as follows, until the handshake is complete and both the client and server have sent their application data to each other:

   a. Call `wrap()` on the client and the server. Check the value of the `SSLEngineResult.HandshakeStatus` field in the `SSLEngineResult` instance that `wrap()` returns:

      - If the handshake isn't complete, then the parameter `dst` will contain handshake data that has to be sent over the network to the peer.
      - If the handshake is complete, then `dst` will contain application data encrypted by `SSLEngine`, ready to be sent to the remote peer.

   b. Add code to handle the `SSLEngineResult.HandshakeStatus` value returned by the `wrap()` and `unwrap()` methods. See Understanding SSLEngine Operation Statuses for more information.

   c. If the `wrap()` method generated data in the network data buffer (which can contain either handshake data or encrypted application data), then send it over the network to the remote peer.

      > **Note**
      >
      > - It is the responsibility of your application, not `SSLEngine`, to send data in the network data buffer to the remote peer.
      > - After you call `wrap()`, you must ensure that all data in the network data buffer has been sent to the peer.
      >
      > For example, Example 8-2 sends network data to the remote peer by calling `SocketChannel.write()`. It checks that all network data has been sent by calling `ByteBuffer.hasRemaining()`:
      >
      > ```
      > while (myNetData.hasRemaining()) {
      >     socketChannel.write(myNetData);
      > }
      > ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

d. Obtain network data sent over the network by the remote peer. Note that it's the responsibility of your application, not `SSLEngine`, to do this. For example, Example 8-2 obtains network data from the remote peer by calling `SocketChannel.read()`:

```
case NEED_UNWRAP:
    // Receive handshaking data from peer
    if (socketChannel.read(peerNetData) < 0) {
        // The channel has reached end-of-stream
    }
```

e. With the network data obtained from the remote peer, call `unwrap()` on the client and the server. Check the value of the `SSLEngineResult.HandshakeStatus` field in the `SSLEngineResult` instance that `unwrap()` returns:

- If the handshake isn't complete, then the `src` parameter may contain additional handshake packets, or more packets will need to be obtained from the peer to continue the handshake.
- If the handshake is complete, then `dst` may contain application data decrypted by `SSLEngine`, ready to be processed by the application.

f. Ensure that the client and server handles the `SSLEngineResult.HandshakeStatus` value returned by `unwrap()`.

Example 8-5 Sample Code for Creating a Nonblocking SocketChannel

The following example is an SSL application that uses a non-blocking `SocketChannel` to communicate with its peer. It sends the string "hello" to the peer by encoding it using the `SSLEngine` created in Example 8-4 . It uses information from the `SSLSession` to determine how large to make the byte buffers.

> **Note**
>
> The example can be made more robust and scalable by using a `Selector` with the nonblocking `SocketChannel`.

```
    // Create a nonblocking socket channel
    SocketChannel socketChannel = SocketChannel.open();
    socketChannel.configureBlocking(false);
    socketChannel.connect(new InetSocketAddress(hostname, port));

    // Complete connection
    while (!socketChannel.finishedConnect()) {
        // do something until connect completed
    }

    // Create byte buffers for holding application and encoded data

    SSLSession session = engine.getSession();
    ByteBuffer myAppData =
ByteBuffer.allocate(session.getApplicationBufferSize());
    ByteBuffer myNetData = ByteBuffer.allocate(session.getPacketBufferSize());
    ByteBuffer peerAppData =
ByteBuffer.allocate(session.getApplicationBufferSize());
    ByteBuffer peerNetData =
ByteBuffer.allocate(session.getPacketBufferSize());
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
// Do initial handshake
doHandshake(socketChannel, engine, myNetData, peerNetData);

myAppData.put("hello".getBytes());
myAppData.flip();

while (myAppData.hasRemaining()) {
    // Generate TLS/DTLS encoded data (handshake or application data)
    SSLEngineResult res = engine.wrap(myAppData, myNetData);

    // Process status of call
    if (res.getStatus() == SSLEngineResult.Status.OK) {
        myAppData.compact();

        // Send TLS/DTLS encoded data to peer
        while(myNetData.hasRemaining()) {
            int num = socketChannel.write(myNetData);
            if (num == 0) {
                // no bytes written; try again later
            }
        }
    }

    // Handle other status:  BUFFER_OVERFLOW, CLOSED
    ...
}
```

Example 8-6 Sample Code for Reading Data From Nonblocking SocketChannel

`SocketChannelSSLEngine`Example 8-4

```
// Read TLS/DTLS encoded data from peer
int num = socketChannel.read(peerNetData);
if (num == -1) {
    // The channel has reached end-of-stream
} else if (num == 0) {
    // No bytes read; try again ...
} else {
    // Process incoming data
    peerNetData.flip();
    res = engine.unwrap(peerNetData, peerAppData);

    if (res.getStatus() == SSLEngineResult.Status.OK) {
        peerNetData.compact();

    if (peerAppData.hasRemaining()) {
        // Use peerAppData
    }
}

// Handle other status:  BUFFER_OVERFLOW, BUFFER_UNDERFLOW, CLOSED
...
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

SSLEngine for DTLS Protocols

This section shows you how to create an `SSLEngine` object and use it to handle a DTLS handshake, generate and process DTLS data, and handle retransmissions in DTLS connections.

Creating an SSLEngine Object for DTLS

The following examples illustrate how to create an `SSLEngine` object for DTLS.

> **Note**
>
> The server name and port number are not used for communicating with the server (all transport is the responsibility of the application). They are hints to the JSSE provider to use for DTLS session caching, and for Kerberos-based cipher suite implementations to determine which server credentials should be obtained.

Example 8-7 Sample Code for Creating an SSLEngine Client for DTLS with PKCS12 as Keystore

The following sample code creates an `SSLEngine` client for DTLS that uses PKCS12 as keystore:

```
import javax.net.ssl.*;
import java.security.*;

// Create and initialize the SSLContext with key material
char[] passphrase = "passphrase".toCharArray();

// First initialize the key and trust material
KeyStore ksKeys = KeyStore.getInstance("PKCS12");
ksKeys.load(new FileInputStream("testKeys"), passphrase);
KeyStore ksTrust = KeyStore.getInstance("PKCS12");
ksTrust.load(new FileInputStream("testTrust"), passphrase);

// KeyManagers decide which key material to use
KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
kmf.init(ksKeys, passphrase);

// TrustManagers decide whether to allow connections
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(ksTrust);

// Get an instance of SSLContext for DTLS protocols
sslContext = SSLContext.getInstance("DTLS");
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

// Create the engine
SSLEngine engine = sslContext.createSSLengine(hostname, port);

// Use engine as client
engine.setUseClientMode(true);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 8-8 Sample Code for Creating an SSLEngine Server for DTLS with PKCS12 as Keystore

`SSLEngine`

```
import javax.net.ssl.*;
import java.security.*;

// Create and initialize the SSLContext with key material
char[] passphrase = "passphrase".toCharArray();

// First initialize the key and trust material
KeyStore ksKeys = KeyStore.getInstance("PKCS12");
ksKeys.load(new FileInputStream("testKeys"), passphrase);
KeyStore ksTrust = KeyStore.getInstance("PKCS12");
ksTrust.load(new FileInputStream("testTrust"), passphrase);

// KeyManagers decide which key material to use
KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
kmf.init(ksKeys, passphrase);

// TrustManagers decide whether to allow connections
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(ksTrust);

// Get an SSLContext for DTLS Protocol without authentication
sslContext = SSLContext.getInstance("DTLS");
sslContext.init(null, null, null);

// Create the engine
SSLEngine engine = sslContext.createSSLeEngine(hostname, port);

// Use the engine as server
engine.setUseClientMode(false);

// Require client authentication
engine.setNeedClientAuth(true);
```

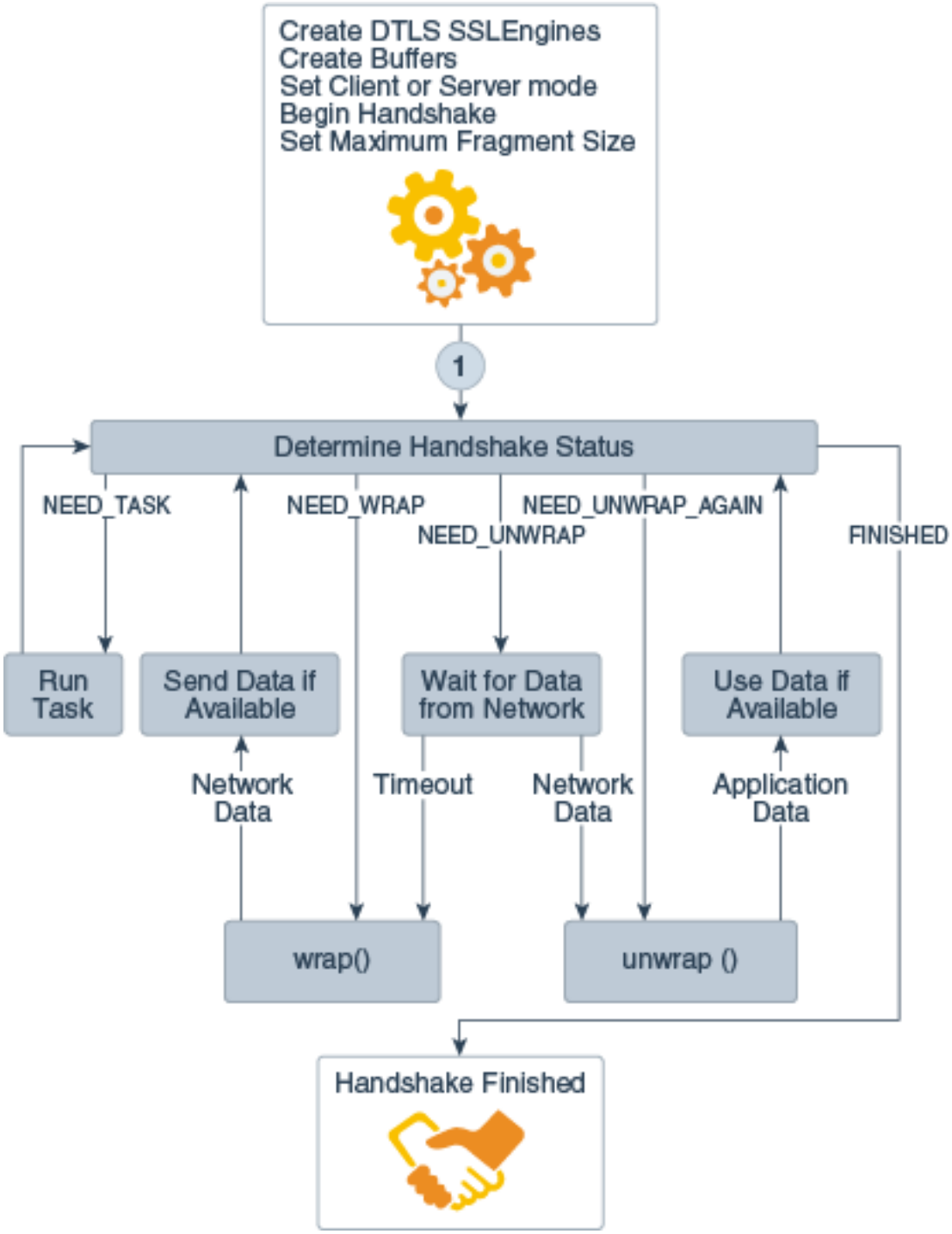
Generating and Processing DTLS Data

A DTLS handshake and a TLS handshake generate and process data similarly. (See Generating and Processing TLS Data.) They both use the `SSLEngine.wrap()` and `SSLEngine.wrap()` methods to generate and consume network data, respectively.

The following diagram shows the state machine during a typical DTLS handshake, with corresponding messages and statuses:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 8-4 State Machine during DTLS Handshake



Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 8-9 Sample Code for Handling DTLS Handshake Status and Overall Status

This sample demonstrates how to handle DTLS handshake status (from the `SSLEngine.getHandshakeStatus` method) and overall status (from the `SSLEngineResult.getStatus` method).

```
void handshake(SSLEngine engine, DatagramSocket socket, SocketAddress
peerAddr) throws Exception {
    boolean endLoops = false;
    // private static int MAX_HANDSHAKE_LOOPS = 60;
    int loops = MAX_HANDSHAKE_LOOPS;
    engine.beginHandshake();
    while (!endLoops && (serverException == null) && (clientException ==
null)) {
        if (--loops < 0) {
            throw new RuntimeException("Too many loops to produce handshake
packets");
        }
        SSLEngineResult.HandshakeStatus hs = engine.getHandshakeStatus();
        if (hs == SSLEngineResult.HandshakeStatus.NEED_UNWRAP ||
                hs == SSLEngineResult.HandshakeStatus.NEED_UNWRAP_AGAIN) {
            ByteBuffer iNet;
            ByteBuffer iApp;
            if (hs == SSLEngineResult.HandshakeStatus.NEED_UNWRAP) {
                // Receive ClientHello request and other SSL/TLS/DTLS records
                byte[] buf = new byte[1024];
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                try {
                    socket.receive(packet);
                } catch (SocketTimeoutException ste) {
                    // Retransmit the packet if timeout
                    List <DatagramPacket> packets = onReceiveTimeout(engine,
peerAddr);
                    for (DatagramPacket p : packets) {
                        socket.send(p);
                    }
                    continue;
                }
                iNet = ByteBuffer.wrap(buf, 0, packet.getLength());
                iApp = ByteBuffer.allocate(1024);
            } else {
                iNet = ByteBuffer.allocate(0);
                iApp = ByteBuffer.allocate(1024);
            }
            SSLEngineResult r = engine.unwrap(iNet, iApp);
            SSLEngineResult.Status rs = r.getStatus();
            hs = r.getHandshakeStatus();
            if (rs == SSLEngineResult.Status.BUFFER_OVERFLOW) {
                // The client maximum fragment size config does not work?
                throw new Exception("Buffer overflow: " +
                                    "incorrect client maximum fragment size");
            } else if (rs == SSLEngineResult.Status.BUFFER_UNDERFLOW) {
                // Bad packet, or the client maximum fragment size
                // config does not work?
                if (hs != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {
                    throw new Exception("Buffer underflow: " +
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                                                "incorrect client maximum fragment
size");
                    } // Otherwise, ignore this packet
                } else if (rs == SSLEngineResult.Status.CLOSED) {
                    endLoops = true;
                } // Otherwise, SSLEngineResult.Status.OK
                if (rs != SSLEngineResult.Status.OK) {
                    continue;
                }
            } else if (hs == SSLEngineResult.HandshakeStatus.NEED_WRAP) {
                // Call a function to produce handshake packets
                List <DatagramPacket> packets = produceHandshakePackets(engine,
peerAddr);
                for (DatagramPacket p : packets) {
                    socket.send(p);
                }
            } else if (hs == SSLEngineResult.HandshakeStatus.NEED_TASK) {
                runDelegatedTasks(engine);
            } else if (hs == SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {
                // OK, time to do application data exchange
                endLoops = true;
            } else if (hs == SSLEngineResult.HandshakeStatus.FINISHED) {
                endLoops = true;
            }
        }
        SSLEngineResult.HandshakeStatus hs = engine.getHandshakeStatus();
        if (hs != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {
            throw new Exception("Not ready for application data yet");
        }
    }
```

Difference Between the TLS and DTLS SSLEngine.wrap() Methods

The `SSLEngine.wrap()` method for DTLS is different from TLS as follows:

- In the TLS implementation of `SSLEngine`, the output buffer of `SSLEngine.wrap()` contains one or more TLS records (due to the TLSv1 BEAST Cipher Block Chaining vulnerability).
- In the DTLS implementation of `SSLEngine`, the output buffer of `SSLEngine.wrap()` contains at most one record, so that every DTLS record can be marshaled and delivered to the datagram layer individually.

> **Note**
>
> Each record produced by `SSLEngine.wrap()` should comply to the maximum packet size limitation as specified by `SSLParameters.getMaximumPacketSize()`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Handling Retransmissions in DTLS Connections

In SSL/TLS over a reliable connection, data is guaranteed to arrive in the proper order, and retransmission is unnecessary. However, for DTLS, which often works over unreliable media, missing or delayed handshake messages must be retransmitted.

The `SSLEngine` class operates in a completely transport-neutral manner, and the application layer performs all I/O. Because the `SSLEngine` class isn't responsible for I/O, the application instead is responsible for providing timers and signalling the `SSLEngine` class when a retransmission is needed. The application layer must determine the right timeout value and when to trigger the timeout event. During handshaking, if an `SSLEngine` object is in `HandshakeStatus.NEED_UNWRAP` state, a call to `SSLEngine.wrap()` means that the previous packets were lost, and must be retransmitted. For such cases, the DTLS implementation of the `SSLEngine` class takes the responsibility to wrap the previous necessary handshaking messages again if necessary.

> **Note**
>
> In a DTLS engine, only handshake messages must be properly exchanged. Application data can handle packet loss without the need for timers.

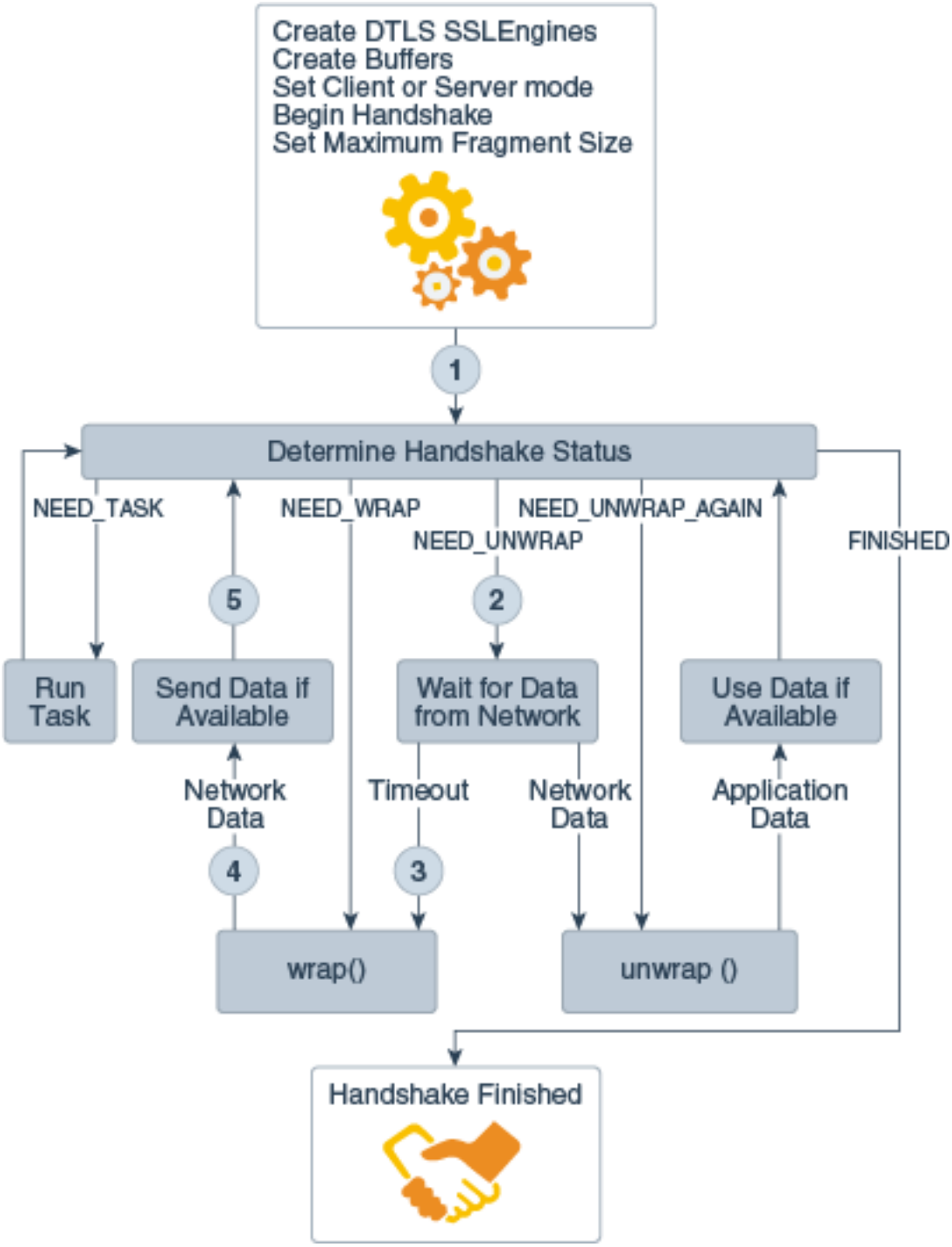
Handling Retransmission in an Application

`SSLEngine.unwrap()` and `SSLEngine.wrap()` can be used together to handle retransmission in an application.

Figure 8-5 shows a typical scenario for handling DTLS handshaking retransmission:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 8-5 DTLS Handshake Retransmission State Flow



1. Create and initialize an instance of DTLS `SSLEngine`.

   See Creating an SSLEngine Object. The DTLS handshake process begins.

2. If the handshake status is `HandshakeStatus.NEED_UNWRAP`, wait for data from network.

3. If the timer times out, it indicates that the previous delivered handshake messages may have been lost.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> In DTLS handshaking retransmission, the determined handshake status isn't necessarily `HandshakeStatus.NEED_WRAP` for the call to `SSLEngine.wrap()`.

4. Call `SSLEngine.wrap()`.
5. The wrapped packets are delivered.

Handling a Buffered Handshake Message in an Application

Datagram transport doesn't require or provide reliable or in-order delivery of data. Handshake messages may be lost or need to be reordered. In the DTLS implementation, a handshake message may need to be buffered for future handling before all previous messages have been received.

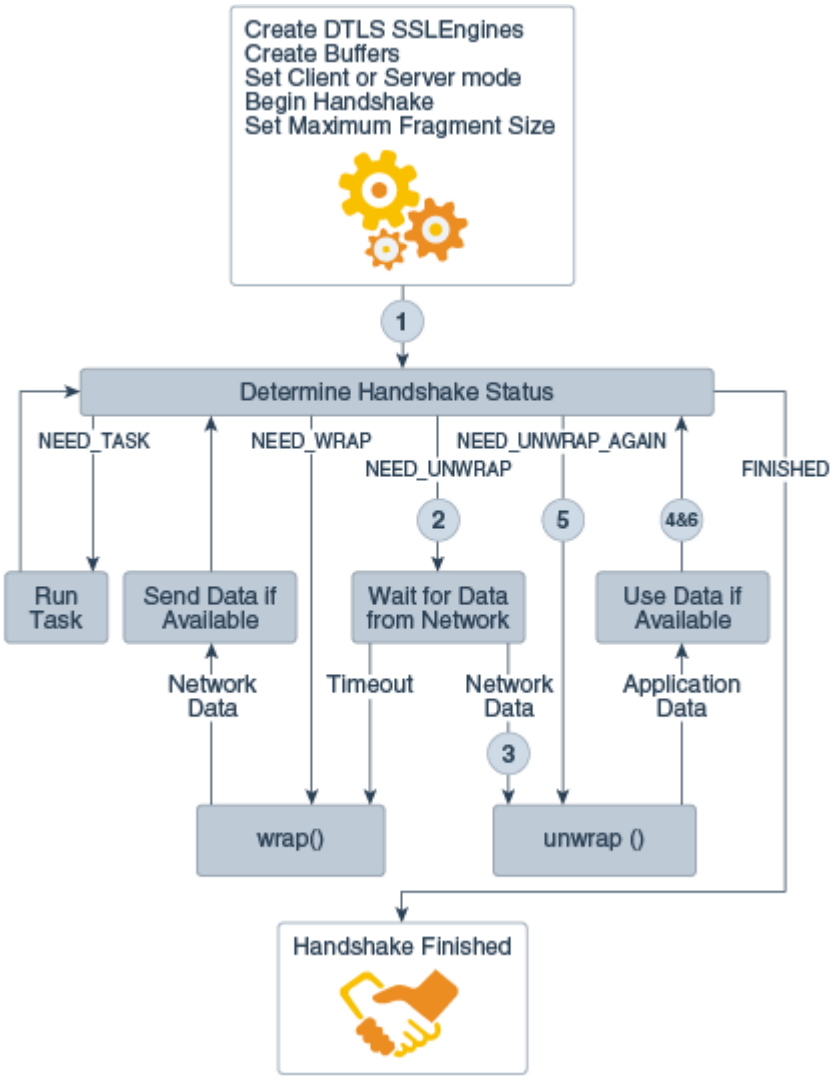
The DTLS implementation of `SSLEngine` takes the responsibility to reorder handshake messages. Handshake message buffering and reordering are transparent to applications.

However, applications must manage `HandshakeStatus.NEED_UNWRAP_AGAIN` status. This status indicates that for the next `SSLEngine.unwrap()` operation no additional data from the remote side is required.

Figure 8-6 shows a typical scenario for using the `HandshakeStatus.NEED_UNWRAP_AGAIN`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 8-6 State Machine of DTLS Buffered Handshake with NEED_UNWRAP_AGAIN



1. Create and initialize an instance of DTLS `SSLEngine`.

   See Creating an SSLEngine Object.

2. Optional: If the handshake status is `HandshakeStatus.NEED_UNWRAP`, wait for data from network.
3. Optional: If you received the network data, call `SSLEngine.unwrap()`.
4. Determine the handshake status for next processing. The handshake status can be `HandshakeStatus.NEED_UNWRAP_AGAIN`, `HandshakeStatus.NEED_UNWRAP`, or `HandshakeStatus.NEED_WRAP`.
   - If the handshake status is `HandshakeStatus.NEED_UNWRAP_AGAIN`, call `SSLEngine.unwrap()`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> For `HandshakeStatus.NEED_UNWRAP_AGAIN` status, no additional data from the network is required for an `SSLEngine.unwrap()` operation.

5. Determine the handshake status for further processing. The handshake status can be `HandshakeStatus.NEED_UNWRAP_AGAIN`, `HandshakeStatus.NEED_UNWRAP`, or `HandshakeStatus.NEED_WRAP`.

Dealing With Blocking Tasks

During handshaking, an `SSLEngine` might encounter tasks that can block or take a long time. For example, a `TrustManager` may need to connect to a remote certificate validation service, or a `KeyManager` might need to prompt a user to determine which certificate to use as part of client authentication. To preserve the nonblocking nature of `SSLEngine`, when the engine encounters such a task, it will return `SSLEngineResult.HandshakeStatus.NEED_TASK`. Upon receiving this status, the application should invoke `SSLEngine.getDelegatedTask()` to get the task, and then, using the threading model appropriate for its requirements, process the task. The application might, for example, obtain threads from a thread pool to process the tasks, while the main thread handles other I/O.

The following code executes each task in a newly created thread:

```
if (res.getHandshakeStatus() == SSLEngineResult.HandshakeStatus.NEED_TASK) {
    Runnable task;
    while ((task = engine.getDelegatedTask()) != null) {
        new Thread(task).start();
    }
}
```

The `SSLEngine` will block future `wrap()` and `unwrap()` calls until all of the outstanding tasks are completed.

Shutting Down a TLS/DTLS Connection

For an orderly shutdown of a TLS/DTLS connection, the TLS/DTLS protocols require transmission of close messages. Therefore, when an application is done with the TLS/DTLS connection, it should first obtain the close messages from the `SSLEngine`, then transmit them to the peer using its transport mechanism, and finally shut down the transport mechanism. Example 8-10 illustrates this.

In addition to an application explicitly closing the `SSLEngine`, the `SSLEngine` might be closed by the peer (via receipt of a close message while it is processing handshake data), or by the `SSLEngine` encountering an error while processing application or handshake data, indicated by throwing an `SSLException`. In such cases, the application should invoke `SSLEngine.wrap()` to get the close message and send it to the peer until `SSLEngine.isOutboundDone()` returns `true` (as shown in Example 8-10), or until the `SSLEngineResult.getStatus()` returns `CLOSED`.

In addition to orderly shutdowns, there can also be unexpected shutdowns when the transport link is severed before close messages are exchanged. In the previous examples, the application might get `-1` or `IOException` when trying to read from the nonblocking `SocketChannel`, or get `IOException` when trying to write to the non-blocking `SocketChannel`. When you get to the end of your input data, you should call `engine.closeInbound()`, which will verify with the `SSLEngine` that the remote peer has closed cleanly from the TLS/DTLS

Copyright © 1993, 2026, Oracle and/or its affiliates.

perspective. Then the application should still try to shut down cleanly by using the procedure in Example 8-10. Obviously, unlike `SSLSocket`, the application using `SSLEngine` must deal with more state transitions, statuses, and programming. See Sample Code Illustrating the Use of an SSLEngine.

Example 8-10 Sample Code for Shutting Down a SSL/TLS/DTLS Connection

The following code sample illustrates how to shut down a TLS/DTLS connection:

```
// Indicate that application is done with engine
engine.closeOutbound();

while (!engine.isOutboundDone()) {
    // Get close message
    SSLEngineResult res = engine.wrap(empty, myNetData);

    // Check res statuses

    // Send close message to peer
    while(myNetData.hasRemaining()) {
        int num = socketChannel.write(myNetData);
        if (num == 0) {
            // no bytes written; try again later
        }
        myNetData().compact();
    }
}

// Close transport
socketChannel.close();
```

SSLSession and ExtendedSSLSession

The `javax.net.ssl.SSLSession` interface represents a security context negotiated between the two peers of an `SSLSocket` or `SSLEngine` connection. After a session has been arranged, it can be shared by future `SSLSocket` or `SSLEngine` objects connected between the same two peers.

In some cases, parameters negotiated during the handshake are needed later in the handshake to make decisions about trust. For example, the list of valid signature algorithms might restrict the certificate types that can be used for authentication. The `SSLSession` can be retrieved *during* the handshake by calling `getHandshakeSession()` on an `SSLSocket` or `SSLEngine`. Implementations of `TrustManager` or `KeyManager` can use the `getHandshakeSession()` method to get information about session parameters to help them make decisions.

A fully initialized `SSLSession` contains the cipher suite that will be used for communications over a secure socket as well as a nonauthoritative hint as to the network address of the remote peer, and management information such as the time of creation and last use. A session also contains a shared master secret negotiated between the peers that is used to create cryptographic keys for encrypting and guaranteeing the integrity of the communications over an `SSLSocket` or `SSLEngine` connection. The value of this master secret is known only to the underlying secure socket implementation and is not exposed through the `SSLSession` API.

`ExtendedSSLSession` extends the `SSLSession` interface to support additional session attributes. The `ExtendedSSLSession` class adds methods that describe the signature algorithms that are

Copyright © 1993, 2026, Oracle and/or its affiliates.

supported by the local implementation and the peer. The `getRequestedServerNames()` method called on an `ExtendedSSLSession` instance is used to obtain a list of `SNIServerName` objects in the requested Server Name Indication (SNI) Extension. The server should use the requested server names to guide its selection of an appropriate authentication certificate, and/or other aspects of the security policy. The client should use the requested server names to guide its endpoint identification of the peer's identity, and/or other aspects of the security policy.

Calls to the `getPacketBufferSize()` and `getApplicationBufferSize()` methods on `SSLSession` are used to determine the appropriate buffer sizes used by `SSLEngine`.

> **Note**
>
> The TLS protocols specify that implementations are to produce packets containing at most 16 kilobytes (KB) of plain text. However, some implementations violate the specification and generate large records up to 32 KB. If the `SSLEngine.unwrap()` code detects large inbound packets, then the buffer sizes returned by `SSLSession` will be updated dynamically. Applications should always check the BUFFER_OVERFLOW and BUFFER_UNDERFLOW statuses and enlarge the corresponding buffers if necessary. See Understanding SSLEngine Operation Statuses. SunJSSE will always send standard compliant 16 KB records and allow incoming 32 KB records. For a workaround, see the System property `jsse.SSLEngine.acceptLargeFragments` in Customizing JSSE.

TLS Keying Material Exporters

TLS keying material exporters enable applications to generate additional application-level keying material from a connection's negotiated TLS session. This enables support for the various exporter algorithms registered with IETF Internet Assigned Numbers Authority (IANA).

For more information about TLS keying material exporters, see RFC 5705: Keying Material Exporters for Transport Layer Security (TLS) for TLSv1 and TLSv1.2 and section 7.5, Exporters in RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3 for TLSv1.3. See the TLS Exporter Labels section of Transport Layer Security (TLS) Parameters for a list of exporter algorithms registered with IANA.

There are two methods for obtaining the key material:

- `ExtendedSSLSession.exportKeyingMaterialData(String label, byte[] context, int length)`: Returns a `byte` array that contains generated Exported Keying Material (EKM).
- `ExtendedSSLSession.exportKeyingMaterialKey(String keyAlg, String label, byte[] context, int length)`: Returns a `SecretKey` of type `keyAlg` that contains generated EKM. See the section `SecretKey` Algorithms in Java Security Standard Algorithm Names for information about standard secret key algorithm names.

Both of these methods have these parameters:

- `String label`: A disambiguating label string
- `byte[] context`: An optional context value represented as a `byte` array
- `int length`: The number of bytes of EKM material needed

Depending on the underlying key derivation mechanism, the key material produced by `exportKeyingMaterialData` may not be extractable or exportable. If this is the case, then use the method `exportingKeyingMaterialKey`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

HttpsURLConnection Class

The `javax.net.ssl.HttpsURLConnection` class extends the `java.net.HttpURLConnection` class and adds support for HTTPS-specific features.

The HTTPS protocol is similar to HTTP, but HTTPS first establishes a secure channel through TLS sockets and then verifies the identity of the peer (see Cipher Suite Choice and Remote Entity Verification) before requesting or receiving data. The `javax.net.ssl.HttpsURLConnection` class extends the `java.net.HttpURLConnection` class and adds support for HTTPS-specific features. To know more about how HTTPS URLs are constructed and used, see the`java.net.URL`, `java.net.URLConnection`, `java.net.HttpURLConnection`, and `javax.net.ssl.HttpsURLConnection` classes.

Upon obtaining an `HttpsURLConnection` instance, you can configure a number of HTTP and HTTPS parameters before actually initiating the network connection via the `URLConnection.connect()` method. Of particular interest are:

- Setting the Assigned SSLSocketFactory
- Setting the Assigned HostnameVerifier

Setting the Assigned SSLSocketFactory

In some situations, it is desirable to specify the `SSLSocketFactory` that an `HttpsURLConnection` instance uses. For example, you might want to tunnel through a proxy type that is not supported by the default implementation. The new `SSLSocketFactory` could return sockets that have already performed all necessary tunneling, thus allowing `HttpsURLConnection` to use additional proxies.

The `HttpsURLConnection` class has a default `SSLSocketFactory` that is assigned when the class is loaded (this is the factory returned by the `SSLSocketFactory.getDefault()` method). Future instances of `HttpsURLConnection` will inherit the current default `SSLSocketFactory` until a new default `SSLSocketFactory` is assigned to the class via the static `HttpsURLConnection.setDefaultSSLSocketFactory()` method. Once an instance of `HttpsURLConnection` has been created, the inherited `SSLSocketFactory` on this instance can be overridden with a call to the `setSSLSocketFactory()` method.

> **Note**
>
> Changing the default static `SSLSocketFactory` has no effect on existing instances of `HttpsURLConnection`. A call to the `setSSLSocketFactory()` method is necessary to change the existing instances.

You can obtain the per-instance or per-class `SSLSocketFactory` by making a call to the `getSSLSocketFactory()` or `getDefaultSSLSocketFactory()` method, respectively.

Setting the Assigned HostnameVerifier

If the host name does not match the host name in the credentials received as part of the TLS handshake, then it is possible that host name spoofing has occurred. If the implementation cannot determine a host name match with reasonable certainty, then the TLS implementation performs a callback to the instance's assigned `HostnameVerifier` for further checking. The

Copyright © 1993, 2026, Oracle and/or its affiliates.

host name verifier can take whatever steps are necessary to make the determination, such as performing host name pattern matching or perhaps opening an interactive dialog box. An unsuccessful verification by the host name verifier closes the connection. For more information regarding host name verification, see RFC 2818: HTTP over TLS.

The `setHostnameVerifier()` and `setDefaultHostnameVerifier()` methods operate in a similar manner to the `setSSLSocketFactory()` and `setDefaultSSLSocketFactory()` methods, in that `HostnameVerifier` objects are assigned on a per-instance and per-class basis, and the current values can be obtained by a call to the `getHostnameVerifier()` or `getDefaultHostnameVerifier()` method.

Support Classes and Interfaces

The classes and interfaces in this section are provided to support the creation and initialization of `SSLContext` objects, which are used to create `SSLSocketFactory`, `SSLServerSocketFactory`, and `SSLEngine` objects. The support classes and interfaces are part of the `javax.net.ssl` package.

Three of the classes described in this section (SSLContext Class, KeyManagerFactory Class, and TrustManagerFactory Class) are *engine classes*. An engine class is an API class for specific algorithms (or protocols, in the case of `SSLContext`), for which implementations may be provided in one or more Cryptographic Service Provider (provider) packages. See JCA Design Principles and Engine Classes and Algorithms.

The SunJSSE provider that comes standard with JSSE provides `SSLContext`, `KeyManagerFactory`, and `TrustManagerFactory` implementations, as well as implementations for engine classes in the standard `java.security` API. Table 8-1 lists implementations supplied by SunJSSE.

Table 8-1 Implementations Supplied by SunJSSE

Engine Class Implemented	Algorithm or Protocol
`KeyStore`	PKCS12
`KeyManagerFactory`	PKIX, SunX509
`TrustManagerFactory`	PKIX (X509 or SunPKIX), SunX509
`SSLContext`	SSLv3[1], TLSv1, TLSv1.1, TLSv1.2, TLSv1.3, DTLSv1.0, DTLSv1.2

[1] Starting with JDK 8u31, the SSLv3 protocol (Secure Socket Layer) has been deactivated and is not available by default. See the `java.security.Security` property `jdk.tls.disabledAlgorithms` in the *<java_home>*`/conf/security/java.security` file. If SSLv3 is absolutely required, the protocol can be reactivated by removing SSLv3 from the `jdk.tls.disabledAlgorithms` property in the `java.security` file or by dynamically setting this Security Property before JSSE is initialized.

SSLContext Class

The `javax.net.ssl.SSLContext` class is an engine class for an implementation of a secure socket protocol. An instance of this class acts as a factory for `SSLSocket`, `SSLServerSocket`, and `SSLEngine`. An `SSLContext` object holds all of the state information shared across all objects created under that context. For example, session state is associated with the `SSLContext` when it is negotiated through the handshake protocol by sockets created by socket factories provided by the context. These cached sessions can be reused and shared by other sockets created under the same context.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Each instance is configured through its `init` method with the keys, certificate chains, and trusted root CA certificates that it needs to perform authentication. This configuration is provided in the form of key and trust managers. These managers provide support for the authentication and key agreement aspects of the cipher suites supported by the context.

Currently, only X.509-based managers are supported.

Obtaining and Initializing the SSLContext Class

The `SSLContext` class is used to create the `SSLSocketFactory` or `SSLServerSocketFactory` class.

There are two ways to obtain and initialize an `SSLContext`:

- The simplest way is to call the static `SSLContext.getDefault` method on either the `SSLSocketFactory` or `SSLServerSocketFactory` class. This method creates a default `SSLContext` with a default `KeyManager`, `TrustManager`, and `SecureRandom` (a secure random number generator). A default `KeyManagerFactory` and `TrustManagerFactory` are used to create the `KeyManager` and `TrustManager`, respectively. The key material used is found in the default keystore and truststore, as determined by system properties described in Customizing the Default Keystores and Truststores, Store Types, and Store Passwords.
- The approach that gives the caller the most control over the behavior of the created context is to call the static method `SSLContext.getDefault` on the `SSLContext` class, and then initialize the context by calling the instance's proper `init()` method. One variant of the `init()` method takes three arguments: an array of `KeyManager` objects, an array of `TrustManager` objects, and a `SecureRandom` object. The `KeyManager` and `TrustManager` objects are created by either implementing the appropriate interfaces or using the `KeyManagerFactory` and `TrustManagerFactory` classes to generate implementations. The `KeyManagerFactory` and `TrustManagerFactory` can then each be initialized with key material contained in the `KeyStore` passed as an argument to the `init()` method of the `TrustManagerFactory` or `KeyManagerFactory` classes. Finally, the `getTrustManagers()` method (in `TrustManagerFactory`) and `getKeyManagers()` method (in `KeyManagerFactory`) can be called to obtain the array of trust managers or key managers, one for each type of trust or key material.

Once a TLS connection is established, an `SSLSession` is created which contains various information, such as identities established and cipher suite used. The `SSLSession` is then used to describe an ongoing relationship and state information between two entities. Each TLS connection involves one session at a time, but that session may be used on many connections between those entities, simultaneously or sequentially.

Creating an SSLContext Object

Like other JCA provider-based engine classes, `SSLContext` objects are created using the `getInstance()` factory methods of the `SSLContext` class. These static methods each return an instance that implements *at least* the requested secure socket protocol. The returned instance may implement other protocols, too. For example, `getInstance("TLSv1")` may return an instance that implements TLSv1, TLSv1.1, and TLSv1.2. The `getSupportedProtocols()` method returns a list of supported protocols when an `SSLSocket`, `SSLServerSocket`, or

Copyright © 1993, 2026, Oracle and/or its affiliates.

SSLEngine is created from this context. You can control which protocols are actually enabled for an SSL connection by using the setEnabledProtocols(String[] protocols) method.

> **Note**
>
> An SSLContext object is automatically created, initialized, and statically assigned to the SSLSocketFactory class when you call the SSLSocketFactory.getDefault() method. Therefore, you do not have to directly create and initialize an SSLContext object (unless you want to override the default behavior).

To create an SSLContext object by calling the getInstance() factory method, you must specify the protocol name. You may also specify which provider you want to supply the implementation of the requested protocol:

- `public static SSLContext getInstance(String protocol);`
- `public static SSLContext getInstance(String protocol, String provider);`
- `public static SSLContext getInstance(String protocol, Provider provider);`

If just a protocol name is specified, then the system will determine whether an implementation of the requested protocol is available in the environment. If there is more than one implementation, then it will determine whether there is a preferred one.

If both a protocol name and a provider are specified, then the system will determine whether an implementation of the requested protocol is in the provider requested. If there is no implementation, an exception will be thrown.

A protocol is a string (such as "TLS") that describes the secure socket protocol desired. Common protocol names for SSLContext objects are defined in Java Security Standard Algorithm Names.

An SSLContext can be obtained as follows:

```
SSLContext sc = SSLContext.getInstance("TLS");
```

A newly created SSLContext should be initialized by calling the init method:

```
public void init(KeyManager[] km, TrustManager[] tm, SecureRandom random);
```

If the KeyManager[] parameter is null, then an empty KeyManager will be defined for this context. If the TrustManager[] parameter is null, then the installed security providers will be searched for the highest-priority implementation of the TrustManagerFactory class (see TrustManagerFactory Class), from which an appropriate TrustManager will be obtained. Likewise, the SecureRandom parameter may be null, in which case a default implementation will be used.

If the internal default context is used, (for example, an SSLContext is created by SSLSocketFactory.getDefault() or SSLServerSocketFactory.getDefault()), then a default KeyManager and TrustManager are created. The default SecureRandom implementation is also chosen.

Copyright © 1993, 2026, Oracle and/or its affiliates.

TrustManager Interface

The primary responsibility of the `TrustManager` is to determine whether the presented authentication credentials should be trusted. If the credentials are not trusted, then the connection will be terminated. To authenticate the remote identity of a secure socket peer, you must initialize an `SSLContext` object with one or more `TrustManager` objects. You must pass one `TrustManager` for each authentication mechanism that is supported. If null is passed into the `SSLContext` initialization, then a trust manager will be created for you. Typically, a single trust manager supports authentication based on X.509 public key certificates (for example, `X509TrustManager`). Some secure socket implementations may also support authentication based on shared secret keys, Kerberos, or other mechanisms.

`TrustManager` objects are created either by a `TrustManagerFactory`, or by providing a concrete implementation of the interface.

TrustManagerFactory Class

The `javax.net.ssl.TrustManagerFactory` is an engine class for a provider-based service that acts as a factory for one or more types of `TrustManager` objects. Because it is provider-based, additional factories can be implemented and configured to provide additional or alternative trust managers that provide more sophisticated services or that implement installation-specific authentication policies.

Creating a TrustManagerFactory

You create an instance of this class in a similar manner to `SSLContext`, except for passing an algorithm name string instead of a protocol name to the `getInstance()` method:

```
TrustManagerFactory tmf = TrustManagerFactory.getInstance(String algorithm);
TrustManagerFactory tmf = TrustManagerFactory.getInstance(String algorithm,
String provider);
TrustManagerFactory tmf = TrustManagerFactory.getInstance(String algorithm,
Provider provider);
```

A sample call is as follows:

```
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX", "SunJSSE");
```

The preceding call creates an instance of the SunJSSE provider's PKIX trust manager factory. This factory can be used to create trust managers that provide X.509 PKIX-based certification path validity checking.

When initializing an `SSLContext`, you can use trust managers created from a trust manager factory, or you can write your own trust manager, for example, using the `CertPath` API. See Java PKI Programmer's Guide. You do not need to use a trust manager factory if you implement a trust manager using the `X509TrustManager` interface.

A newly created factory should be initialized by calling one of the `init()` methods:

```
public void init(KeyStore ks);
public void init(ManagerFactoryParameters spec);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Call whichever `init()` method is appropriate for the `TrustManagerFactory` you are using. If you are not sure, then ask the provider vendor.

For many factories, such as the SunX509 `TrustManagerFactory` from the SunJSSE provider, the `KeyStore` is the only information required to initialize the `TrustManagerFactory` and thus the first `init` method is the appropriate one to call. The `TrustManagerFactory` will query the `KeyStore` for information about which remote certificates should be trusted during authorization checks.

Sometimes, initialization parameters other than a `KeyStore` are needed by a provider. Users of that provider are expected to pass an implementation of the appropriate `ManagerFactoryParameters` as defined by the provider. The provider can then call the specified methods in the `ManagerFactoryParameters` implementation to obtain the needed information.

For example, suppose the `TrustManagerFactory` provider requires initialization parameters B, R, and S from any application that wants to use that provider. Like all providers that require initialization parameters other than a `KeyStore`, the provider requires the application to provide an instance of a class that implements a particular `ManagerFactoryParameters` subinterface. In the example, suppose that the provider requires the calling application to implement and create an instance of `MyTrustManagerFactoryParams` and pass it to the second `init()` method. The following example illustrates what `MyTrustManagerFactoryParams` can look like:

```
public interface MyTrustManagerFactoryParams extends ManagerFactoryParameters
{
  public boolean getBValue();
  public float getRValue();
  public String getSValue();
}
```

Some trust managers can make trust decisions without being explicitly initialized with a `KeyStore` object or any other parameters. For example, they may access trust material from a local directory service via LDAP, use a remote online certificate status checking server, or access default trust material from a standard local location.

Check TrustManagerFactory Object's Certificates' Expiration Date

The following method, `filterTrustAnchors`, filters a set of trust anchors, removing those whose certificates will expire by the specified date, then initializes a `TrustManagerFactory` object with this set.

```
    public static void filterTrustAnchors (
        String truststore, String password, String validityDate)
        throws FileNotFoundException, KeyStoreException, IOException,
            ParseException, NoSuchAlgorithmException,
            InvalidAlgorithmParameterException, CertificateException,
            KeyManagementException {

        FileInputStream is = new FileInputStream(truststore);
        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
        keystore.load(is, password.toCharArray());
        PKIXParameters params = new PKIXParameters(keystore);

        // Obtain CA root certificates
        Set<TrustAnchor> myTrustAnchors = params.getTrustAnchors();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            // Create new set of CA certificates that are still valid for
            // specified date
            Set<TrustAnchor> validTrustAnchors =
                myTrustAnchors.stream().filter(
                    ta -> {
                        try {
                            ta.getTrustedCert().checkValidity(
                                DateFormat.getDateInstance().parse(validityDate));
                        } catch (CertificateException | ParseException e) {
                            return false;
                        }
                        return true; }).collect(Collectors.toSet());

            // Create PKIXBuilderParameters parameters
            PKIXBuilderParameters pkixParams =
                new PKIXBuilderParameters(validTrustAnchors, new
X509CertSelector());

            // Wrap PKIX parameters as trust manager parameters
            ManagerFactoryParameters trustParams =
                new CertPathTrustManagerParameters(pkixParams);

            // Create TrustManagerFactory for PKIX-compliant trust managers
            TrustManagerFactory factory = TrustManagerFactory.getInstance("PKIX");

            // Pass parameters to factory to be passed to CertPath implementation
            factory.init(trustParams);

            // Use factory
            SSLContext ctx = SSLContext.getInstance("TLS");
            ctx.init(null, factory.getTrustManagers(), null);
        }
```

PKIX TrustManager Support

The default trust manager algorithm is PKIX. It can be changed by editing the `ssl.TrustManagerFactory.algorithm` property in the `java.security` file.

The PKIX trust manager factory uses the CertPath PKIX implementation (see PKI Programmer's Guide Overview) from an installed security provider. The trust manager factory can be initialized using the normal `init(KeyStores)` method, or by passing `CertPath` parameters to the PKIX trust manager using the `CertPathTrustManagerParameters` class.

Example 8-11 illustrates how to get the trust manager to use a particular LDAP certificate store and enable revocation checking.

If the `TrustManagerFactory.init(KeyStore)` method is used, then default PKIX parameters are used with the exception that revocation checking is disabled. It can be enabled by setting the `com.sun.net.ssl.checkRevocation` system property to `true`. This setting requires that the CertPath implementation can locate revocation information by itself. The PKIX implementation in the provider can do this in many cases but requires that the system property `com.sun.security.enableCRLDP` be set to `true`. Note that the `TrustManagerFactory.init(ManagerFactoryParameters)` method has revocation checking enabled by default.

See PKIX Classes and The CertPath Class.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 8-11 Sample Code for Using a LDAP Certificate to Enable Revocation Checking

The following example illustrates how to get the trust manager to use a particular LDAP certificate store and enable revocation checking:

```
    import javax.net.ssl.*;
    import java.security.cert.*;
    import java.security.KeyStore;
    import java.io.FileInputStream;
    ...

    // Obtain Keystore password
    char[] pass = System.console().readPassword("Password: ");

    // Create PKIX parameters
    KeyStore anchors = KeyStore.getInstance("PKCS12");
    anchors.load(new FileInputStream(anchorsFile, pass));
    PKIXBuilderParameters pkixParams = new PKIXBuilderParameters(anchors, new
X509CertSelector());

    // Specify LDAP certificate store to use
    LDAPCertStoreParameters lcsp = new
LDAPCertStoreParameters("ldap.imc.org", 389);
    pkixParams.addCertStore(CertStore.getInstance("LDAP", lcsp));

    // Specify that revocation checking is to be enabled
    pkixParams.setRevocationEnabled(true);

    // Wrap PKIX parameters as trust manager parameters
    ManagerFactoryParameters trustParams = new
CertPathTrustManagerParameters(pkixParams);

    // Create TrustManagerFactory for PKIX-compliant trust managers
    TrustManagerFactory factory = TrustManagerFactory.getInstance("PKIX");

    // Pass parameters to factory to be passed to CertPath implementation
    factory.init(trustParams);

    // Use factory
    SSLContext ctx = SSLContext.getInstance("TLS");
    ctx.init(null, factory.getTrustManagers(), null);
```

X509TrustManager Interface

The `javax.net.ssl.X509TrustManager` interface extends the general `TrustManager` interface. It must be implemented by a trust manager when using X.509-based authentication.

To support X.509 authentication of remote socket peers through JSSE, an instance of this interface must be passed to the `init` method of an `SSLContext` object.

Creating an X509TrustManager

You can either implement this interface directly yourself or obtain one from a provider-based `TrustManagerFactory` (such as that supplied by the SunJSSE provider). You could also

Copyright © 1993, 2026, Oracle and/or its affiliates.

implement your own interface that delegates to a factory-generated trust manager. For example, you might do this to filter the resulting trust decisions and query an end-user through a graphical user interface.

If a null KeyStore parameter is passed to the SunJSSE PKIX or SunX509 `TrustManagerFactory`, then the factory uses the following process to try to find trust material:

1. If the `javax.net.ssl.trustStore` property is defined, then the `TrustManagerFactory` attempts to find a file using the file name specified by that system property, and uses that file for the KeyStore parameter. If the `javax.net.ssl.trustStorePassword` system property is also defined, then its value is used to check the integrity of the data in the truststore before opening it.

   If the `javax.net.ssl.trustStore` property is defined but the specified file does not exist, then a default `TrustManager` using an empty keystore is created.

2. If the `javax.net.ssl.trustStore` system property was not specified, then:
   - if the file *java-home*`/lib/security/jssecacerts` exists, that file is used;
   - if the file *java-home*`/lib/security/cacerts` exists, that file is used;
   - if neither of these files exists, then the TLS cipher suite is anonymous, does not perform any authentication, and thus does not need a truststore.

To know more about what *java-home* refers to, see Terms and Definitions.

The factory looks for a file specified via the `javax.net.ssl.trustStore` Security Property or for the `jssecacerts` file before checking for a `cacerts` file. Therefore, you can provide a JSSE-specific set of trusted root certificates separate from ones that might be present in `cacerts` for code-signing purposes.

Creating Your Own X509TrustManager

If the supplied `X509TrustManager` behavior is not suitable for your situation, then you can create your own `X509TrustManager` by either creating and registering your own `TrustManagerFactory` or by implementing the `X509TrustManager` interface directly.

Example 8-12 illustrates a `MyX509TrustManager` class that enhances the default SunJSSE `X509TrustManager` behavior by providing alternative authentication logic when the default `X509TrustManager` fails.

Once you have created such a trust manager, assign it to an `SSLContext` via the `init()` method, as in the following example. Future `SocketFactories` created from this `SSLContext` will use your new `TrustManager` when making trust decisions.

```
TrustManager[] myTMs = new TrustManager[] { new MyX509TrustManager() };
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(null, myTMs, null);
```

Example 8-12 Sample Code for Creating a X509TrustManager

The following code sample illustrates `MyX509TrustManager` class that enhances the default SunJSSE `X509TrustManager` behavior by providing alternative authentication logic when the default `X509TrustManager` fails:

```
class MyX509TrustManager implements X509TrustManager {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        /*
         * The default PKIX X509TrustManager9.  Decisions are delegated
         * to it, and a fall back to the logic in this class is performed
         * if the default X509TrustManager does not trust it.
         */
        X509TrustManager pkixTrustManager;

        MyX509TrustManager() throws Exception {
            // create a "default" JSSE X509TrustManager.

            KeyStore ks = KeyStore.getInstance("PKCS12");
            ks.load(new FileInputStream("trustedCerts"),
    "passphrase".toCharArray());

            TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
            tmf.init(ks);

            TrustManager tms [] = tmf.getTrustManagers();

            /*
             * Iterate over the returned trust managers, looking
             * for an instance of X509TrustManager.  If found,
             * use that as the default trust manager.
             */
            for (int i = 0; i < tms.length; i++) {
                if (tms[i] instanceof X509TrustManager) {
                    pkixTrustManager = (X509TrustManager) tms[i];
                    return;
                }
            }

            /*
             * Find some other way to initialize, or else the
             * constructor fails.
             */
            throw new Exception("Couldn't initialize");
        }

        /*
         * Delegate to the default trust manager.
         */
        public void checkClientTrusted(X509Certificate[] chain, String authType)
                    throws CertificateException {
            try {
                pkixTrustManager.checkClientTrusted(chain, authType);
            } catch (CertificateException excep) {
                // do any special handling here, or rethrow exception.
            }
        }

        /*
         * Delegate to the default trust manager.
         */
        public void checkServerTrusted(X509Certificate[] chain, String authType)
                    throws CertificateException {
            try {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                pkixTrustManager.checkServerTrusted(chain, authType);
            } catch (CertificateException excep) {
                /*
                 * Possibly pop up a dialog box asking whether to trust the
                 * cert chain.
                 */
            }
        }

        /*
         * Merely pass this through.
         */
        public X509Certificate[] getAcceptedIssuers() {
            return pkixTrustManager.getAcceptedIssuers();
        }
    }
```

Updating the Keystore Dynamically

You can enhance `MyX509TrustManager` to handle dynamic keystore updates. When a `checkClientTrusted` or `checkServerTrusted` test fails and does not establish a trusted certificate chain, you can add the required trusted certificate to the keystore. You must create a new `pkixTrustManager` from the `TrustManagerFactory` initialized with the updated keystore. When you establish a new connection (using the previously initialized `SSLContext`), the newly added certificate will be used when making trust decisions.

X509ExtendedTrustManager Class

The `X509ExtendedTrustManager` class is an abstract implementation of the `X509TrustManager` interface. It adds methods for connection-sensitive trust management. In addition, it enables endpoint verification at the TLS layer.

In TLS 1.2 and later, both client and server can specify which hash and signature algorithms they will accept. To authenticate the remote side, authentication decisions must be based on both X509 certificates and the local accepted hash and signature algorithms. The local accepted hash and signature algorithms can be obtained using the `ExtendedSSLSession.getLocalSupportedSignatureAlgorithms()` method.

The `ExtendedSSLSession` object can be retrieved by calling the `SSLSocket.getHandshakeSession()` method or the `SSLEngine.getHandshakeSession()` method.

The `X509TrustManager` interface is not connection-sensitive. It provides no way to access `SSLSocket` or `SSLEngine` session properties.

Besides TLS 1.2 and later support, the `X509ExtendedTrustManager` class also supports algorithm constraints and SSL layer host name verification. For JSSE providers and trust manager implementations, the `X509ExtendedTrustManager` class is highly recommended over the legacy `X509TrustManager` interface.

Creating an X509ExtendedTrustManager

You can either create an `X509ExtendedTrustManager` subclass yourself (which is outlined in the following section) or obtain one from a provider-based `TrustManagerFactory` (such as that supplied by the SunJSSE provider). In Java SE 7, the PKIX or SunX509 `TrustManagerFactory` returns an `X509ExtendedTrustManager` instance.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Creating Your Own X509ExtendedTrustManager

This section outlines how to create a subclass of `X509ExtendedTrustManager` in nearly the same way as described for `X509TrustManager`.

Example 8-13 illustrates how to create a class that uses the PKIX `TrustManagerFactory` to locate a default `X509ExtendedTrustManager` that will be used to make decisions about trust.

Example 8-13 Sample Code for Creating a PKIX TrustManagerFactory

The following code sample illustrates how to create a class that uses the PKIX `TrustManagerFactory` to locate a default `X509ExtendedTrustManager` that will be used to make decisions about trust. If the default trust manager fails for any reason, then the subclass can add other behavior. In the sample, these locations are indicated by comments in the `catch` clauses.

```
import java.io.*;
import java.net.*;
import java.security.*;
import java.security.cert.*;
import javax.net.ssl.*;

public class MyX509ExtendedTrustManager extends X509ExtendedTrustManager {

  /*
   * The default PKIX X509ExtendedTrustManager.  Decisions are
   * delegated to it, and a fall back to the logic in this class is
   * performed if the default X509ExtendedTrustManager does not
   * trust it.
   */

  X509ExtendedTrustManager pkixTrustManager;

  MyX509ExtendedTrustManager() throws Exception {
    // create a "default" JSSE X509ExtendedTrustManager.

    KeyStore ks = KeyStore.getInstance("PKCS12");
    ks.load(new FileInputStream("trustedCerts"), "passphrase".toCharArray());

    TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
    tmf.init(ks);

    TrustManager tms [] = tmf.getTrustManagers();

    /*
     * Iterate over the returned trust managers, looking
     * for an instance of X509ExtendedTrustManager. If found,
     * use that as the default trust manager.
     */
    for (int i = 0; i < tms.length; i++) {
      if (tms[i] instanceof X509ExtendedTrustManager) {
        pkixTrustManager = (X509ExtendedTrustManager) tms[i];
        return;
      }
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        /*
         * Find some other way to initialize, or else we have to fail the
         * constructor.
         */
        throw new Exception("Couldn't initialize");
    }

    /*
     * Delegate to the default trust manager.
     */
    public void checkClientTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {
        try {
            pkixTrustManager.checkClientTrusted(chain, authType);
        } catch (CertificateException excep) {
            // do any special handling here, or rethrow exception.
        }
    }

    /*
     * Delegate to the default trust manager.
     */
    public void checkServerTrusted(X509Certificate[] chain, String authType)
        throws CertificateException {
        try {
            pkixTrustManager.checkServerTrusted(chain, authType);
        } catch (CertificateException excep) {
            /*
             * Possibly pop up a dialog box asking whether to trust the
             * cert chain.
             */
        }
    }

    /*
     * Connection-sensitive verification.
     */
    public void checkClientTrusted(X509Certificate[] chain, String authType,
SSLSocket socket)
        throws CertificateException {
        try {
            pkixTrustManager.checkClientTrusted(chain, authType, socket);
        } catch (CertificateException excep) {
            // do any special handling here, or rethrow exception.
        }
    }

    public void checkClientTrusted(X509Certificate[] chain, String authType,
SSLEngine engine)
        throws CertificateException {
        try {
            pkixTrustManager.checkClientTrusted(chain, authType, engine);
        } catch (CertificateException excep) {
            // do any special handling here, or rethrow exception.
        }
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
  public void checkServerTrusted(X509Certificate[] chain, String authType,
Socket socket)
    throws CertificateException {
    try {
      pkixTrustManager.checkServerTrusted(chain, authType, socket);
    } catch (CertificateException excep) {
      // do any special handling here, or rethrow exception.
    }
  }

  public void checkServerTrusted(X509Certificate[] chain, String authType,
SSLEngine engine)
    throws CertificateException {
    try {
      pkixTrustManager.checkServerTrusted(chain, authType, engine);
    } catch (CertificateException excep) {
      // do any special handling here, or rethrow exception.
    }
  }

  /*
   * Merely pass this through.
   */
  public X509Certificate[] getAcceptedIssuers() {
    return pkixTrustManager.getAcceptedIssuers();
  }
}
```

KeyManager Interface

The primary responsibility of the `KeyManager` is to select the authentication credentials that will eventually be sent to the remote host. To authenticate yourself (a local secure socket peer) to a remote peer, you must initialize an `SSLContext` object with one or more `KeyManager` objects. You must pass one `KeyManager` for each different authentication mechanism that will be supported. If null is passed into the `SSLContext` initialization, then an empty `KeyManager` will be created. If the internal default context is used (for example, an `SSLContext` created by `SSLSocketFactory.getDefault()` or `SSLServerSocketFactory.getDefault()`), then a default `KeyManager` is created. See Customizing the Default Keystores and Truststores, Store Types, and Store Passwords. Typically, a single key manager supports authentication based on X.509 public key certificates. Some secure socket implementations may also support authentication based on shared secret keys, Kerberos, or other mechanisms.

`KeyManager` objects are created either by a `KeyManagerFactory`, or by providing a concrete implementation of the interface.

KeyManagerFactory Class

The `javax.net.ssl.KeyManagerFactory` class is an engine class for a provider-based service that acts as a factory for one or more types of `KeyManager` objects. The SunJSSE provider implements a factory that can return a basic X.509 key manager. Because it is provider-based, additional factories can be implemented and configured to provide additional or alternative key managers.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Creating a KeyManagerFactory

You create an instance of this class in a similar manner to `SSLContext`, except for passing an algorithm name string instead of a protocol name to the `getInstance()` method:

```
KeyManagerFactory kmf = getInstance(String algorithm);
KeyManagerFactory kmf = getInstance(String algorithm, String provider);
KeyManagerFactory kmf = getInstance(String algorithm, Provider provider);
```

A sample call as follows:

```
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509", "SunJSSE");
```

The preceding call creates an instance of the SunJSSE provider's default key manager factory, which provides basic X.509-based authentication keys.

A newly created factory should be initialized by calling one of the `init` methods:

```
public void init(KeyStore ks, char[] password);
public void init(ManagerFactoryParameters spec);
```

Call whichever `init` method is appropriate for the `KeyManagerFactory` you are using. If you are not sure, then ask the provider vendor.

For many factories, such as the default SunX509 `KeyManagerFactory` from the SunJSSE provider, the `KeyStore` and password are the only information required to initialize the `KeyManagerFactory` and thus the first `init` method is the appropriate one to call. The `KeyManagerFactory` will query the `KeyStore` for information about which private key and matching public key certificates should be used for authenticating to a remote socket peer. The password parameter specifies the password that will be used with the methods for accessing keys from the `KeyStore`. All keys in the `KeyStore` must be protected by the same password.

Sometimes initialization parameters other than a `KeyStore` and password are needed by a provider. Users of that provider are expected to pass an implementation of the appropriate `ManagerFactoryParameters` as defined by the provider. The provider can then call the specified methods in the `ManagerFactoryParameters` implementation to obtain the needed information.

Some factories can provide access to authentication material without being initialized with a `KeyStore` object or any other parameters. For example, they may access key material as part of a login mechanism such as one based on JAAS, the Java Authentication and Authorization Service.

As previously indicated, the SunJSSE provider supports a SunX509 factory that must be initialized with a `KeyStore` parameter.

X509KeyManager Interface

The `javax.net.ssl.X509KeyManager` interface extends the general `KeyManager` interface. It must be implemented by a key manager for X.509-based authentication. To support X.509 authentication to remote socket peers through JSSE, an instance of this interface must be passed to the `init()` method of an `SSLContext` object.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Creating an X509KeyManager

You can either implement this interface directly yourself or obtain one from a provider-based `KeyManagerFactory` (such as that supplied by the SunJSSE provider). You could also implement your own interface that delegates to a factory-generated key manager. For example, you might do this to filter the resulting keys and query an end-user through a graphical user interface.

Creating Your Own X509KeyManager

If the default `X509KeyManager` behavior is not suitable for your situation, then you can create your own `X509KeyManager` in a way similar to that shown in Creating Your Own X509TrustManager.

X509ExtendedKeyManager Class

The `X509ExtendedKeyManager` abstract class is an implementation of the `X509KeyManager` interface that allows for connection-specific key selection. It adds two methods that select a key alias for client or server based on the key type, allowed issuers, and current `SSLEngine`:

- `public String chooseEngineClientAlias(String[] keyType, Principal[] issuers, SSLEngine engine)`
- `public String chooseEngineServerAlias(String keyType, Principal[] issuers, SSLEngine engine)`

If a key manager is not an instance of the `X509ExtendedKeyManager` class, then it will not work with the `SSLEngine` class.

For JSSE providers and key manager implementations, the `X509ExtendedKeyManager` class is highly recommended over the legacy `X509KeyManager` interface.

In TLS 1.2 and later, both client and server can specify which hash and signature algorithms they will accept. To pass the authentication required by the remote side, local key selection decisions must be based on both X509 certificates and the remote accepted hash and signature algorithms. The remote accepted hash and signature algorithms can be retrieved using the `ExtendedSSLSession.getPeerSupportedSignatureAlgorithms()` method.

You can create your own `X509ExtendedKeyManager` subclass in a way similar to that shown in Creating Your Own X509TrustManager.

Support for the Server Name Indication (SNI) Extension on the server side enables the key manager to check the server name and select the appropriate key accordingly. For example, suppose there are three key entries with certificates in the keystore:

- `cn=www.example.com`
- `cn=www.example.org`
- `cn=www.example.net`

If the ClientHello message requests to connect to `www.example.net` in the SNI extension, then the server should be able to select the certificate with subject `cn=www.example.net`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Relationship Between a TrustManager and a KeyManager

Historically, there has been confusion regarding the functionality of a `TrustManager` and a `KeyManager`.

A `TrustManager` determines whether the remote authentication credentials (and thus the connection) should be trusted.

A `KeyManager` determines which authentication credentials to send to the remote host.

Secondary Support Classes and Interfaces

These classes are provided as part of the JSSE API to support the creation, use, and management of secure sockets. They are less likely to be used by secure socket applications than are the core and support classes. The secondary support classes and interfaces are part of the `javax.net.ssl` and `javax.security.cert` packages.

SSLParameters Class

The `SSLParameters` class encapsulates the following parameters that affect a SSL/TLS/DTLS connection:

- The list of cipher suites to be accepted in a TLS/DTLS handshake
- The list of signature schemes to be accepted in a TLS/DTLS handshake
- The list of protocols to be allowed
- The endpoint identification algorithm during TLS/DTLS handshaking
- The server names and server name matchers (see Server Name Indication (SNI) Extension)
- The cipher suite preference to be used in a TLS/DTLS handshake
- Algorithm during TLS/DTLS handshaking
- The Server Name Indication (SNI)
- The maximum network packet size
- The algorithm constraints and whether TLS/DTLS servers should request or require client authentication

You can retrieve the current `SSLParameters` for an `SSLSocket` or `SSLEngine` by using the following methods:

- `getSSLParameters()` in an `SSLSocket`, `SSLServerSocket`, and `SSLEngine`
- `getDefaultSSLParameters()` and `getSupportedSSLParamters()` in an `SSLContext`

You can assign `SSLParameters` with the `setSSLParameters()` method in an `SSLSocket`, `SSLServerSocket` and `SSLEngine`.

You can explicitly set the server name indication with the `SSLParameters.setServerNames()` method. The server name indication in client mode also affects endpoint identification. In the implementation of `X509ExtendedTrustManager`, it uses the server name indication retrieved by the `ExtendedSSLSession.getRequestedServerNames()` method. See Example 8-14.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 8-14 Sample Code to Set Server Name Indication

This example uses the host name in the server name indication (`www.example.com`) to make endpoint identification against the peer's identity presented in the end-entity's X.509 certificate.

```
SSLSocketFactory factory = ...
SSLSocket sslSocket = factory.createSocket("172.16.10.6", 443);
// SSLEngine sslEngine = sslContext.createSSLEngine("172.16.10.6", 443);

SNIHostName serverName = new SNIHostName("www.example.com");
List<SNIServerName> serverNames = new ArrayList<>(1);
serverNames.add(serverName);

SSLParameters params = sslSocket.getSSLParameters();
params.setServerNames(serverNames);
sslSocket.setSSLParameters(params);
// sslEngine.setSSLParameters(params);
```

Cipher Suite Preference

During TLS handshaking, the client requests to negotiate a cipher suite from a list of cryptographic options that it supports, starting with its first preference. Then, the server selects a single cipher suite from the list of cipher suites requested by the client. The selection honors the server's preference by default, which is the most secure setting. However, the server can choose to honor the client's preference rather than its own preference by invoking the method `SSLParameters.setUseCipherSuitesOrder(false)`.

SSLSessionContext Interface

The `javax.net.ssl.SSLSessionContext` interface is a grouping of SSLSession objects associated with a single entity. For example, it could be associated with a server or client that participates in many sessions concurrently. The methods in this interface enable the enumeration of all sessions in a context and allow lookup of specific sessions via their session IDs.

An `SSLSessionContext` may optionally be obtained from an `SSLSession` by calling the SSLSession `getSessionContext()` method. The context may be unavailable in some environments, in which case the `getSessionContext()` method returns null.

SSLSessionBindingListener Interface

The `javax.net.ssl.SSLSessionBindingListener` interface is implemented by objects that are notified when they are being bound or unbound from an SSLSession.

SSLSessionBindingEvent Class

The `javax.net.ssl.SSLSessionBindingEvent` class defines the event communicated to an `SSLSessionBindingListener` (see SSLSessionBindingListener Interface) when it is bound or unbound from an `SSLSession` (see SSLSession and ExtendedSSLSession).

Copyright © 1993, 2026, Oracle and/or its affiliates.

HandShakeCompletedListener Interface

The `javax.net.ssl.HandShakeCompletedListener` interface is an interface implemented by any class that is notified of the completion of an SSL protocol handshake on a given `SSLSocket` connection.

HandShakeCompletedEvent Class

The `javax.net.ssl.HandShakeCompletedEvent` class defines the event communicated to a `HandShakeCompletedListener` (see HandShakeCompletedListener Interface) upon completion of an SSL protocol handshake on a given `SSLSocket` connection.

HostnameVerifier Interface

If the SSL/TLS implementation's standard host name verification logic fails, then the implementation calls the `verify()` method of the class that implements this interface and is assigned to this `HttpsURLConnection` instance. If the callback class can determine that the host name is acceptable given the parameters, it reports that the connection should be allowed. An unacceptable response causes the connection to be terminated. See Example 8-15.

See `HttpsURLConnection` for more information about how to assign the `HostnameVerifier` to the `HttpsURLConnection`.

Example 8-15 Sample Code for Implementing the HostnameVerifier Interface

The following example illustrates a class that implements `HostnameVerifier` interface:

```
public class MyHostnameVerifier implements HostnameVerifier {

    public boolean verify(String hostname, SSLSession session) {
        // pop up an interactive dialog box
        // or insert additional matching logic
        if (good_address) {
            return true;
        } else {
            return false;
        }
    }
}

//...deleted...

HttpsURLConnection urlc = (HttpsURLConnection)
  (new URL("https://www.example.com/")).openConnection();
urlc.setHostnameVerifier(new MyHostnameVerifier());
```

X509Certificate Class

Many secure socket protocols perform authentication using public key certificates, also called X.509 certificates. This is the default authentication mechanism for the TLS protocol.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The `java.security.cert.X509Certificate` abstract class provides a standard way to access the attributes of X.509 certificates.

> **Note**
>
> The `javax.security.cert.X509Certificate` class is supported only for backward compatibility with previous (1.0.x and 1.1.x) versions of JSSE. New applications should use the `java.security.cert.X509Certificate` class instead.

AlgorithmConstraints Interface

The `java.security.AlgorithmConstraints` interface is used for controlling allowed cryptographic algorithms. `AlgorithmConstraints` defines three `permits()` methods. These methods tell whether an algorithm name or a key is permitted for certain cryptographic functions. Cryptographic functions are represented by a set of `CryptoPrimitive`, which is an enumeration containing fields like `STREAM_CIPHER`, `MESSAGE_DIGEST`, and `SIGNATURE`.

Thus, an `AlgorithmConstraints` implementation can answer questions like: Can I use this key with this algorithm for the purpose of a cryptographic operation?

An `AlgorithmConstraints` object can be associated with an `SSLParameters` object by using the new `setAlgorithmConstraints()` method. The current `AlgorithmConstraints` object for an `SSLParameters` object is retrieved using the `getAlgorithmConstraints()` method.

StandardConstants Class

The `StandardConstants` class is used to represent standard constants definitions in JSSE.

`StandardConstants.SNI_HOST_NAME` represents a domain name server (DNS) host name in a Server Name Indication (SNI) extension, which can be used when instantiating an `SNIServerName` or `SNIMatcher` object.

SNIServerName Class

An instance of the abstract `SNIServerName` class represents a server name in the Server Name Indication (SNI) extension. It is instantiated using the type and encoded value of the specified server name.

You can use the `getType()` and `getEncoded()` methods to return the server name type and a copy of the encoded server name value, respectively. The `equals()` method can be used to check if some other object is "equal" to this server name. The `hashCode()` method returns a hash code value for this server name. To get a string representation of the server name (including the server name type and encoded server name value), use the `toString()` method.

SNIMatcher Class

An instance of the abstract `SNIMatcher` class performs match operations on an `SNIServerName` object. Servers can use information from the Server Name Indication (SNI) extension to decide if a specific `SSLSocket` or `SSLEngine` should accept a connection. For example, when multiple "virtual" or "name-based" servers are hosted on a single underlying network address, the server application can use SNI information to determine whether this server is the exact server that the client wants to access. Instances of this class can be used by a server to verify the acceptable server names of a particular type, such as host names.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The `SNIMatcher` class is instantiated using the specified server name type on which match operations will be performed. To match a given `SNIServerName`, use the `matches()` method. To return the server name type of the given `SNIMatcher` object, use the `getType()` method.

SNIHostName Class

An instance of the `SNIHostName` class (which extends the `SNIServerName` class) represents a server name of type "host_name" (see StandardConstants Class) in the Server Name Indication (SNI) Extension. To instantiate an `SNIHostName`, specify the fully qualified DNS host name of the server (as understood by the client) as a `String` argument. The argument is illegal in the following cases:

- The argument is empty.
- The argument ends with a trailing period.
- The argument is not a valid Internationalized Domain Name (IDN) compliant with the RFC 3490 specification.

You can also instantiate an `SNIHostName` by specifying the encoded host name value as a byte array. This method is typically used to parse the encoded name value in a requested SNI extension. Otherwise, use the `SNIHostName(String hostname)` constructor. The `encoded` argument is illegal in the following cases:

- The argument is empty.
- The argument ends with a trailing period.
- The argument is not a valid Internationalized Domain Name (IDN) compliant with the RFC 3490 specification.
- The argument is not encoded in UTF-8 or US-ASCII.

> **Note**
>
> The `encoded` byte array passed in as an argument is cloned to protect against subsequent modification.

To return the host name of an `SNIHostName` object in US-ASCII encoding, use the `getAsciiName()` method. To compare a server name to another object, use the `equals()` method (comparison is *not* case-sensitive). To return a hash code value of an `SNIHostName`, use the `hashCode()` method. To return a string representation of an `SNIHostName`, including the DNS host name, use the `toString()` method.

You can create an `SNIMatcher` object for an `SNIHostName` object by passing a regular expression representing one or more host names to match to the `createSNIMatcher()` method.

Customizing JSSE

JSSE includes a standard implementation that can be customized by plugging in different implementations or specifying the default keystore, and so on.

Table 8-2 and Table 8-3 summarize which aspects can be customized, what the defaults are, and which mechanisms are used to provide customization.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Some of the customizations are done by setting system property or Security Property values. Sections following the table explain how to set such property values.

> **Note**
>
> Many of the properties shown in this table are currently used by the JSSE implementation, but there is no guarantee that they will continue to have the same names and types (system or security) or even that they will exist at all in future releases. All such properties are flagged with an asterisk (*). They are documented here for your convenience for use with the JSSE implementation.

Table 8-2 shows items that are customized by setting the `java.security.Security` property. See How to Specify a java.security.Security Property

Table 8-2 Security Properties and Customized Items

Security Property	Customized Item	Default Value	Notes
`cert.provider.x509v1`	Customizing the X509Certificate Implementation	X509Certificate implementation from Oracle	None
JCE encryption algorithms used by the SunJSSE provider	Give alternative JCE algorithm providers a higher preference order than the SunJCE provider	SunJCE implementations	None
`jdk.certpath.disabledAlgorithms*`	Disabled certificate verification cryptographic algorithms (see Disabled and Restricted Cryptographic Algorithms)	MD2, MD5, SHA1 jdkCA & usage TLSServer, RSA keySize < 1024, DSA keySize < 1024, EC keySize < 224, SHA1 usage SignedJAR & denyAfter 2019-01-01[1]	None
`jdk.crypto.disabledAlgorithms*`	Disabled and Restricted Cryptographic Algorithms	None	You can override the value of this Security Property by setting the value of the system property of the same name. This enables you to re-enable algorithms for applications that require them.
`jdk.tls.disabledAlgorithms*`	Disabled and Restricted Cryptographic Algorithms	SSLv3, TLSv1, TLSv1.1, DTLSv1.0, RC4, DES, MD5withRSA, DH keySize < 1024, EC keySize < 224, 3DES_EDE_CBC, anon, NULL, ECDH, TLS_RSA_*, rsa_pkcs1_sha1 usage HandshakeSignature, ecdsa_sha1 usage HandshakeSignature, dsa_sha1 usage HandshakeSignature[1]	Disables specific algorithms (protocols versions, cipher suites, key exchange mechanisms, etc.) that will not be negotiated for TLS/DTLS connections, even if they are enabled explicitly in an application

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-2 (Cont.) Security Properties and Customized Items

Security Property	Customized Item	Default Value	Notes
`jdk.tls.keyLimits*`	Limiting Amount of Data Algorithms May Encrypt with a Set of Keys	AES/GCM/NoPadding KeyUpdate 2^37, ChaCha20-Poly1305 KeyUpdate 2^3	Limits the amount of data an algorithm may encrypt with a specific set of keys; once this limit is reached, a KeyUpdate post-handshake message is sent, which requests that the current set of keys be updated.
`jdk.tls.legacyAlgorithms*`	Legacy Cryptographic Algorithms	NULL, anon, RC4, DES, 3DES_EDE_CBC[1]	Specifies which algorithms are considered legacy algorithms, which are not negotiated during TLS/DTLS security parameters negotiation unless there are no other candidates.
`jdk.tls.server.defaultDHEParameters`	Diffie-Hellman groups	Safe prime Diffie-Hellman groups in OpenJDK TLS/DTLS implementation	Defines default finite field Diffie-Hellman ephemeral (DHE) parameters for (Datagram) Transport Layer Security ((D)TLS) processing
`ocsp.enable*`	Client-Driven OCSP and OCSP Stapling	`false`	Enables client-driven Online Certificate Status Protocol (OCSP). You must also enable revocation checking; see Setting up a Java Client to use Client-Driven OCSP.
`security.provider.`*n*	Cryptographic service provider; see Customizing the Provider Implementation and Customizing the Encryption Algorithm Providers	The first five providers in order of priority are: 1. SUN 2. SunRsaSign 3. SunEC 4. SunJSSE 5. SunJCE	Specify the provider in the `security.provider.`*n*= line in security properties file, where *n* is an integer whose value is equal or greater than 1.
`ssl.KeyManagerFactory.algorithm`	Default key manager factory algorithm name (see Customizing the Default Key Managers and Trust Managers)	SunX509	None
`ssl.ServerSocketFactory.provider*`	Default `SSLServerSocketFactory` implementation	`SSLServerSocketFactory` implementation from Oracle	None
`ssl.SocketFactory.provider*`	Default `SSLSocketFactory` implementation	`SSLSocketFactory` implementation from Oracle	None
`ssl.TrustManagerFactory.algorithm`	Default trust manager factory algorithm name (see Customizing the Default Key Managers and Trust Managers)	PKIX	None

Copyright © 1993, 2026, Oracle and/or its affiliates.

[1] The list of restricted, disabled, and legacy algorithms specified in these Security Properties may change; see the `java.security` file in your JDK installation for the latest values.

* This Security Property is currently used by the JSSE implementation, but it is not guaranteed to be examined and used by other implementations. If it is examined by another implementation, then that implementation should handle it in the same manner as the JSSE implementation does. There is no guarantee the property will continue to exist or be of the same type (system or security) in future releases.

Table 8-3 shows items that are customized by setting `java.lang.System` property. See How to Specify a java.lang.System Property.

Table 8-3 System Properties and Customized Items

System Property	Customized Item	Default	Notes
`com.sun.net.ssl.checkRevocation*`	Revocation checking	`false`	You must enable revocation checking to enable client-driven OCSP; see Client-Driven OCSP and OCSP Stapling.
Customize via `port` field in the HTTPS URL.*	Default HTTPS port	443	None
`https.cipherSuites*`	Default cipher suites for HTTPS connections	Determined by the socket factory.	This contains a comma-separated list of cipher suite names specifying which cipher suites to enable for use on this `HttpsURLConnection`. See the `SSLSocket.setEnabledCipherSuites(String[])` method. Note that this method sets the preference order of the ClientHello cipher suites directly from the `String` array passed to it.
`https.protocols*`	Default handshaking protocols for HTTPS connections	Determined by the socket factory.	This contains a comma-separated list of protocol suite names specifying which protocol suites to enable on this `HttpsURLConnection`. See `SSLSocket.setEnabledProtocols(String[])`
`https.proxyHost*`	Default proxy host	None	None
`https.proxyPort*`	Default proxy port	80	None
`java.protocol.handler.pkgs`	Specifying an Alternative HTTPS Protocol Implementation	Implementation from Oracle	None
`javax.net.ssl.keyStore*`	Default keystore; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	The value `NONE` may be specified. This setting is appropriate if the keystore is not file-based (for example, it resides in a hardware token)

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`javax.net.ssl.keyStorePassword*`	Default keystore password; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	It is inadvisable to specify the password in a way that exposes it to discovery by other users. For example, specifying the password on the command line. To keep the password secure, have the application prompt for the password, or specify the password in a properly protected option file
`javax.net.ssl.keyStoreProvider*`	Default keystore provider; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	None
`javax.net.ssl.keyStoreType*`	Default keystore type; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	`KeyStore.getDefaultType()`	None
`javax.net.ssl.sessionCacheSize`	Default value for the maximum number of entries in the SSL session cache	`20480`	The session cache size can be set by calling the `SSLSessionContext.setSessionCacheSize` method or by setting the `javax.net.ssl.sessionCacheSize` system property. If the cache size is not set, the default value is used.
`javax.net.ssl.trustStore*`	Default truststore; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	`jssecacerts`, if it exists. Otherwise, `cacerts`	The value `NONE` may be specified. This setting is appropriate if the truststore is not file-based (for example, it resides in a hardware token).
`javax.net.ssl.trustStorePassword*`	Default truststore password; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	It is inadvisable to specify the password in a way that exposes it to discovery by other users. For example, specifying the password on the command line. To keep the password secure, have the application prompt for the password, or specify the password in a properly protected option file
`javax.net.ssl.trustStoreProvider*`	Default truststore provider; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	None

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`javax.net.ssl.trustStoreType*`	Default truststore type; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	`KeyStore.getDefaultType()`	None
`jdk.crypto.disabledAlgorithms*`	Disabled and Restricted Cryptographic Algorithms	None	You can override the value of the `jdk.crypto.disabledAlgorithms` Security Property by setting the value of this system property. This enables you to re-enable algorithms for applications that require them.
`jdk.tls.acknowledgeCloseNotify*`	Specifying That close_notify Alert Is Sent When One Is Received	`false`	If `true`, then when the client or server receives a close_notify alert, it sends a corresponding close_notify alert and the connection is duplex closed.
`jdk.tls.allowLegacyResumption`	Support for the TLS Session Hash and Extended Master Secret Extension	`true`	None
`jdk.tls.allowLegacyMasterSecret`	Support for the TLS Session Hash and Extended Master Secret Extension	`true`	None
`jdk.tls.allowUnsafeServerCertChange*`	Defines whether unsafe server certificate change in an SSL/TLS renegotiation should be restricted or not; see Allow Unsafe Server Certificate Change in SSL/TLS Renegotiations	`false`	**Caution**: Do not set this system property to `true` unless it is really necessary as this would re-establish the unsafe server certificate change vulnerability.
`jdk.tls.client.cipherSuites*`	Client-side default enabled cipher suites; see Specifying Default Enabled Cipher Suites	See SunJSSE Cipher Suites for a list of currently implemented SunJSSE cipher suites for this JDK release, sorted by order of preference	**Caution**: These system properties can be used to configure weak cipher suites, or the configured cipher suites may be weak in the future. It is not recommended that you use these system properties without understanding the risks.
`jdk.tls.client.disableExtensions*`	Configuring Default Extensions	None	Blocks extensions used on the client side.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`jdk.tls.client.enableCAExtension*`	Enabling certificate_authorities Extension for Server Certificate Selection	`false`	If the client or server trusts more CAs such that it exceeds the size limit of the extension (less than 2^16 bytes), then the extension is not enabled. Also, some server implementations don't allow handshake messages to exceed 2^14 bytes. Thus, there may be interoperability issues if `jdk.tls.client.enableCAExtension` is set to `true` and the client trusts more CAs such that it exceeds the server implementation limit.
`jdk.tls.client.enableSessionTicketExtension*`	Resuming Session Without Server-Side State	`true`	If `true`, the client will send a session ticket extension in the ClientHello for TLS 1.2 and earlier. This extension enables the client to accept the server's session state for server-side stateless TLS session resumption (RFC 5077).
`jdk.tls.client.maxInboundCertificateChainLength*`	Setting the Maximum Certificate Chain Length Accepted from the Client or Server During TLS/DTLS Handshaking	None	If you set the system property `jdk.tls.client.maxInboundCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum length of a server certificate chain accepted by a client. Otherwise, if you have set the system property `jdk.tls.maxCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum server certificate chain length. If you haven't set either property, then the default value 10 is the maximum server certificate chain length.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`jdk.tls.client.protocols*`	Default handshaking protocols for TLS/DTLS clients. See The SunJSSE Provider	None	To enable specific SunJSSE protocols on the client, specify them in a comma-separated list within quotation marks; all other supported protocols are not enabled on the client For example, • If `jdk.tls.client.protocols="TLSv1,TLSv1.1"`, then the default protocol settings on the client for TLSv1 and TLSv1.1 are enabled, while SSLv3, TLSv1.2, TLSv1.3, and SSLv2Hello are not enabled • If `jdk.tls.client.protocols="DTLSv1.2"`, then the protocol setting on the client for DTLS1.2 is enabled, while DTLS1.0 is not enabled
`jdk.tls.client.SignatureSchemes*`	Specifying Signature Schemes That Can Be Used over the TLS/DTLS Protocols.	If this system property is not defined or the value is empty, then the implementation default signature schemes and preferences will be used.	This contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS connections on the client side. For example: `jdk.tls.client.SignatureSchemes="ecdsa_secp521r1_sha512,rsa_pkcs1_sha512,rsa_pss_pss_sha512"` Unrecognized or unsupported signature scheme names specified in the property are ignored. The names are not case sensitive. For a list of signature scheme names, see the section Signature Schemes in *Java Security Standard Algorithm Names Specification*.
`jdk.tls.ephemeralDHKeySize*`	Customizing the Size of Ephemeral Diffie-Hellman Keys	2048 bits	None

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`jdk.tls.maxCertificateChainLength*`	Setting the Maximum Certificate Chain Length Accepted from the Client or Server During TLS/DTLS Handshaking	• 8: Maximum client certificate chain length accepted by a server • 10: Maximum server certificate chain accepted by a client	Specifies the maximum allowed length of the certificate chain in TLS/DTLS handshaking. A service can function as both a TLS/DTLS server and client. When the service acts as a server, it enforces a maximum certificate chain length accepted from clients. When the service acts as a client, it enforces a maximum certificate chain length accepted from servers. If the system property `jdk.tls.server.maxInboundCertificateChainLength` or `jdk.tls.client.maxInboundCertificateChainLength` is set to a value equal to or greater than 0, then this value overrides the value of `jdk.tls.maxCertificateChainLength`.
`jdk.tls.maxHandshakeMessageSize*`	Setting the Maximum Certificate Chain Length Accepted from the Client or Server During TLS/DTLS Handshaking	32768 (32 kilobytes)	Specifies the maximum allowed size, in bytes, for the handshake message in TLS/DTLS handshaking.
`jdk.tls.namedGroups*`	Customizing the Supported Named Groups for TLS/DTLS Key Exchange	If this system property is not defined or the value is empty, then the implementation default groups and preferences will be used.	This contains a comma-separated list within quotation marks of enabled named groups in preference order. For example: `jdk.tls.namedGroups="X25519MLKEM768,secp521r1,secp256r1,ffdhe2048"` The standard list of key exchange named groups are defined in the Named Groups section of *Java Security Standard Algorithm Names Specification*.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`jdk.tls.rejectClientInitiatedRenegotiation*`	Rejects client-initiated renegotiation on the server side. If this system property is `true`, then the server will not accept client initiated renegotiations and will fail with a fatal `handshake_failure` alert. Rejects server-side client-initialized renegotiation.	`false`	None
`jdk.tls.server.cipherSuites*`	Server-side default enabled cipher suites. See Specifying Default Enabled Cipher Suites	See SunJSSE Cipher Suites to determine which cipher suites are enabled by default	**Caution**: These system properties can be used to configure weak cipher suites, or the configured cipher suites may be weak in the future. It is not recommended that you use these system properties without understanding the risks.
`jdk.tls.server.disableExtensions*`	Configuring Default Extensions	None	Blocks extensions used on the server side.
`jdk.tls.server.enableSessionTicketExtension*`	Resuming Session Without Server-Side State	true	If `true`, the server will provide stateless session tickets, if the client supports it, as described in RFC 5077 (TLS Session Resumption Without Server-Side State) for TLS 1.2 and earlier and RFC 8446 for TLS 1.3. A stateless session ticket contains the encrypted server's state, which saves server resources.
`jdk.tls.server.maxInboundCertificateChainLength*`	Setting the Maximum Certificate Chain Length Accepted from the Client or Server During TLS/DTLS Handshaking	None	If you set the system property `jdk.tls.server.maxInboundCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum length of a client certificate chain accepted by a server. Otherwise, if you have set the system property `jdk.tls.maxCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum client certificate chain length. If you haven't set either property, then the default value 8 is the maximum client certificate chain length.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`jdk.tls.server.newSessionTicket*`	Specifies the number of TLSv1.3 resumption tickets sent by a JSSE server per session.	1	The value of this property is an integer between 0 and 10.
`jdk.tls.server.protocols*`	Default handshaking protocols for TLS/DTLS servers. See The SunJSSE Provider	None	To configure the default enabled protocol suite on the server side of a SunJSSE provider, specify the protocols in a comma-separated list within quotation marks. The protocols in this list are standard SSL protocol names as described in Java Security Standard Algorithm Names. Note that this System Property impacts only the default protocol suite (SSLContext of the algorithms SSL, TLS and DTLS). If an application uses a version-specific SSLContext (SSLv3, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3, DTLSv1.0, or DTLSv1.2), or sets the enabled protocol version explicitly, this System Property has no impact.
`jdk.tls.server.sessionTicketTimeout*`	Specifies how long a session in the server cache or stateless resumption tickets are available for use	86400 seconds (24 hours)	You can modify the value set with this property during run time with the method `SSLSessionContext.setSessionTimeout()`.
`jdk.tls.server.SignatureSchemes*`	Specifying Signature Schemes That Can Be Used over the TLS/DTLS Protocols.	If this system property is not defined or empty, then the provider-specific default is used.	This is a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS connections on the server side. For example: `jdk.tls.server.SignatureSchemes="ecdsa_secp521r1_sha512,rsa_pkcs1_sha512,rsa_pss_pss_sha512"` Unrecognized or unsupported signature scheme names specified in the property are ignored. The names are not case sensitive. For a list of signature scheme names, see the section Signature Schemes in *Java Security Standard Algorithm Names Specification*.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`jdk.tls.useExtendedMasterSecret`	Support for the TLS Session Hash and Extended Master Secret Extension	`true`	None
`jsse.enableFFDHE*`	Enables or disables Finite Field Diffie-Hellman Ephemeral (FFDHE) parameters for TLS/DTLS key exchange	`true`	FFDHE is a TLS/DTLS extension defined in RFC 7919. It enables TLS/DTLS connections to use known finite field Diffie-Hellman groups. Some very old TLS vendors may not be able handle TLS extensions. In this case, set this property to `false` to disable the FFDHE extension.
`jsse.enableMFLNExtension*`	Customizing the Maximum Fragment Length Negotiation (MFLN) Extension	`false`	None
`jsse.enableSNIExtension*`	Server Name Indication option	`true`	Server Name Indication (SNI) is a TLS extension, defined in RFC 6066. It enables TLS connections to virtual servers, in which multiple servers for different network names are hosted at a single underlying network address. Some very old TLS vendors may not be able handle TLS extensions. In this case, set this property to `false` to disable the SNI extension
`jsse.SSLEngine.acceptLargeFragments*`	Default sizing buffers for large TLS packets	None	Setting this system property to `true`, `SSLSession` will size buffers to handle large data packets by default. This may cause applications to allocate unnecessarily large `SSLEngine` buffers. Instead, applications should dynamically check for buffer overflow conditions and resize buffers as appropriate
`sun.security.ssl.allowLegacyHelloMessages*`	Allows legacy Hello messages; see SunJSSE Renegotiation Interoperability Modes	`true`	Setting this system property to `true` allows the peer to handshake without requiring the proper RFC 5746 messages. This system property is deprecated and might be removed in a future JDK release.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-3 (Cont.) System Properties and Customized Items

System Property	Customized Item	Default	Notes
`sun.security.ssl.allowUnsafeRenegotiation*`	Allows unsafe SSL/TLS renegotiations; see SunJSSE Renegotiation Interoperability Modes	`false`	Setting this system property to `true` permits full (unsafe) legacy renegotiation. This system property is deprecated and might be removed in a future JDK release.

* This system property is currently used by the JSSE implementation, but it is not guaranteed to be examined and used by other implementations. If it is examined by another implementation, then that implementation should handle it in the same manner as the JSSE implementation does. There is no guarantee the property will continue to exist or be of the same type (system or security) in future releases.

How to Specify a java.lang.System Property

You can customize some aspects of JSSE by setting system properties. There are several ways to set these properties:

- To set a system property statically, use the `-D` option of the `java` command. For example, to run an application named MyApp and set the `javax.net.ssl.trustStore` system property to specify a truststore named MyCacertsFile. See truststore. Enter the following:

```
java -Djavax.net.ssl.trustStore=MyCacertsFile MyApp
```

- To set a system property dynamically, call the `java.lang.System.setProperty()` method in your code:

```
System.setProperty("propertyName", "propertyValue");
```

For example, a `setProperty()` call corresponding to the previous example for setting the `javax.net.ssl.trustStore` system property to specify a truststore named "`MyCacertsFile`" would be:

```
System.setProperty("javax.net.ssl.trustStore", "MyCacertsFile");
```

How to Specify a java.security.Security Property

You can customize some aspects of JSSE by setting Security Properties. You can set a Security Property either statically or dynamically:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- To set a Security Property statically, add a line to the security properties file. The security properties file is located at *java-home*`/conf/security/java.security`

  java-home
  See Terms and Definitions

  To specify a Security Property value in the security properties file, you add a line of the following form:

  ```
  propertyName=propertyValue
  ```

  For example, suppose that you want to specify a different key manager factory algorithm name than the default SunX509. You do this by specifying the algorithm name as the value of a Security Property named `ssl.KeyManagerFactory.algorithm`. For example, to set the value to `MyKeyFactoryAlgorithm`, add the following line to the security properties file:

  ```
  ssl.KeyManagerFactory.algorithm=MyKeyFactoryAlgorithm
  ```

  > **Note**
  >
  > Properties in the `java.security` file are typically parsed only once. If you have modified any property in this file, restart your applications to ensure that the changes are properly reflected.

- To set a Security Property dynamically, call the `java.security.Security.setProperty` method in your code:

  ```
  Security.setProperty("propertyName," "propertyValue");
  ```

  For example, a call to the `setProperty()` method corresponding to the previous example for specifying the key manager factory algorithm name would be:

  ```
  Security.setProperty("ssl.KeyManagerFactory.algorithm", "MyX509");
  ```

Customizing the X509Certificate Implementation

The X509Certificate implementation returned by the `X509Certificate.getInstance()` method is by default the implementation from the JSSE implementation.

To cause a different implementation to be returned:

Specify the name (and package) of the other implementation's class as the value of a How to Specify a java.security.Security Property named `cert.provider.x509v1`.

`MyX509CertificateImplcom.cryptox`

```
cert.provider.x509v1=com.cryptox.MyX509CertificateImpl
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Specifying Default Enabled Cipher Suites

You can specify the default enabled cipher suites in your application or with the system properties `jdk.tls.client.cipherSuites` and `jdk.tls.server.cipherSuites`.

> **Note**
>
> The actual use of enabled cipher suites is restricted by algorithm constraints.

The set of cipher suites to enable by default is determined by one of the following ways in this order of preference:

1. Explicitly set by application
2. Specified by system property
3. Specified by JSSE provider defaults

For example, explicitly setting the default enabled cipher suites in your application overrides settings specified in `jdk.tls.client.cipherSuites` or `jdk.tls.server.cipherSuites` as well as JSSE provider defaults.

Explicitly Set by Application

You can set which cipher suites are enabled with one of the following methods:

- `SSLSocket.setEnabledCipherSuites(String[])`
- `SSLEngine.setEnabledCipherSuites(String[])`
- `SSLServerSocket.setEnabledCipherSuites(String[])`
- `SSLParameters(String[] cipherSuites)`
- `SSLParameters(String[] cipherSuites, String[] protocols)`
- `SSLParameters.setCipherSuites(String[])`
- `https.cipherSuites` system property for `HttpsURLConnection`

Specified by System Property

The system property `jdk.tls.client.cipherSuites` specifies the default enabled cipher suites on the client side; `jdk.tls.server.cipherSuites` specifies those on the server side.

The syntax of the value of these two system properties is a comma-separated list of supported cipher suite names. Unrecognized or unsupported cipher suite names that are specified in these properties are ignored. See Java Security Standard Algorithms for standard JSSE cipher suite names.

> **Note**
>
> These system properties are currently supported by Oracle JDK and OpenJDK. They are not guaranteed to be supported by other JDK implementations.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **⚠ Caution**
>
> These system properties can be used to configure weak cipher suites, or the configured cipher suites may be weak in the future. It is not recommended that you use these system properties without understanding the risks.

Specified by JSSE Provider Defaults

Each JSSE provider has its own default enabled cipher suites. See The SunJSSE Provider in JDK Providers Documentation for the cipher suite names supported by the SunJSSE provider and which ones that are enabled by default.

Specifying an Alternative HTTPS Protocol Implementation

You can communicate securely with an SSL-enabled web server by using the HTTPS URL scheme for the `java.net.URL` class. The JDK provides a default HTTPS URL implementation.

If you want an alternative HTTPS protocol implementation to be used, set the `java.protocol.handler.pkgs` How to Specify a java.lang.System Property to include the new class name. This action causes the specified classes to be found and loaded before the JDK default classes. See the URL class for details.

Customizing the Provider Implementation

The JDK comes with a JSSE Cryptographic Service Provider, or *provider* for short, named SunJSSE. Providers are essentially packages that implement one or more engine classes for specific cryptographic algorithms.

The JSSE engine classes are `SSLContext`, `KeyManagerFactory`, and `TrustManagerFactory`. See Java Cryptography Architecture (JCA) Reference Guide to know more about providers and engine classes.

Before it can be used, a provider must be registered, either statically or dynamically. You do not need to register the SunJSSE provider because it is preregistered. If you want to use other providers, read the following sections to see how to register them.

Registering the Cryptographic Provider Statically

Register a provider statically by adding a line of the following form to the security properties file, `<java-home>/conf/security/java.security`:

```
security.provider.n=provName|className
```

This declares a provider, and specifies its preference order *n*. The preference order is the order in which providers are searched for requested algorithms when no specific provider is requested. The order is 1-based; 1 is the most preferred, followed by 2, and so on.

provName is the provider's name and *className* is the fully qualified class name of the provider.

Standard security providers are automatically registered for you in the `java.security` security properties file.

To use another JSSE provider, add a line registering the other provider, giving it whatever preference order you prefer.

Copyright © 1993, 2026, Oracle and/or its affiliates.

You can have more than one JSSE provider registered at the same time. The registered providers may include different implementations for different algorithms for different engine classes, or they may have support for some or all of the same types of algorithms and engine classes. When a particular engine class implementation for a particular algorithm is searched for, if no specific provider is specified for the search, then the providers are searched in preference order and the implementation from the first provider that supplies an implementation for the specified algorithm is used.

See Step 8.1: Configure the Provider in Steps to Implement and Integrate a Provider.

Registering the Cryptographic Service Provider Dynamically

Instead of registering a provider statically, you can add the provider dynamically at runtime by calling either the `addProvider` or `insertProviderAt` method in the `Security` class.

See Step 8.1: Configure the Provider in Steps to Implement and Integrate a Provider.

Provider Configuration

Some providers may require configuration. This is done using the `configure` method of the `Provider` class, prior to calling the `addProvider` method of the `Security` class. See SunPKCS11 Configuration for an example. The `Provider.configure()` method is new to Java SE 9.

Configuring the Preferred Provider for Specific Algorithms

Specify the preferred provider for a specific algorithm in the `jdk.security.provider.preferred` Security Property. By specifying a preferred provider you can configure providers that offer performance gains for specific algorithms but are not the best performing provider for other algorithms. The ordered provider list specified using the `security.provider.n` property is not sufficient to order providers that offer performance gains for specific algorithms but are not the best performing provider for other algorithms. More flexibility is required for configuring the ordering of provider list to achieve performance gains.

The `jdk.security.provider.preferred` Security Property allows specific algorithms, or service types to be selected from a preferred set of providers before accessing the list of registered providers. See How to Specify a java.security.Security Property.

The `jdk.security.provider.preferred` Security Property does not register the providers. The ordered provider list must be Registering the Cryptographic Provider Statically using the `security.provider.n` property. Any provider that is not registered is ignored.

Specifying the Preferred Provider for an Algorithm

The syntax for specifying the preferred providers string in the `jdk.security.provider.preferred` Security Property is a comma-separated list of `ServiceType.Algorithm:Provider`

In this syntax:

ServiceType
The name of the service type (for example: `"MessageDigest"`). ServiceType is optional. If it isn't specified, then the algorithm applies to all service types.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Algorithm
The standard algorithm name. See Java Security Standard Algorithm Names. Algorithms can be specified as full standard name, (AES/CBC/PKCS5Padding) or as partial (AES, AES/CBC, AES//PKCS5Padding).

Provider
The name of the provider. Any provider that isn't listed in the registered list is ignored. See JDK Providers.

Entries containing errors such as parsing errors are ignored. Use the command `java -Djava.security.debug=jca` to debug errors.

Preferred Providers and FIPS

If you add a FIPS provider to the `security.provider.n` property, and specify the preferred provider ordering in the `jdk.security.provider.preferred` property then the preferred providers specified in `jdk.security.provider.preferred` are selected first.

Hence, it is recommended that you don't configure `jdk.security.provider.preferred` property for FIPS provider configurations.

jdk.security.provider.preferred Default Values

The `jdk.security.provider.preferred` property is not set by default and is used only for application performance tuning.

Example 8-16 Sample jdk.security.provider.preferred Property

The syntax for specifying the `jdk.security.provider.preferred` property is as follows:

```
jdk.security.provider.preferred=AES/GCM/NoPadding:SunJCE,
MessageDigest.SHA-256:SUN
```

In this syntax:

ServiceType
MessageDigest

Algorithm
AES/GCM/NoPadding, SHA-256

Provider
SunJCE, SUN

Customizing the Default Keystores and Truststores, Store Types, and Store Passwords

Whenever a default `SSLSocketFactory` or `SSLServerSocketFactory` is created (via a call to `SSLSocketFactory.getDefault` or `SSLServerSocketFactory.getDefault`), and this default `SSLSocketFactory` (or `SSLServerSocketFactory`) comes from the JSSE reference implementation, a default `SSLContext` is associated with the socket factory. (The default socket factory will come from the JSSE implementation.)

This default `SSLContext` is initialized with a default `KeyManager` and a default `TrustManager`. If a keystore is specified by the `javax.net.ssl.keyStore` system property and an appropriate `javax.net.ssl.keyStorePassword` system property (see How to Specify a java.lang.System Property), then the `KeyManager` created by the default `SSLContext` will be a `KeyManager`

Copyright © 1993, 2026, Oracle and/or its affiliates.

implementation for managing the specified keystore. (The actual implementation will be as specified in Customizing the Default Key Managers and Trust Managers.) If no such system property is specified, then the keystore managed by the `KeyManager` will be a new empty keystore.

Generally, the peer acting as the server in the handshake will need a keystore for its KeyManager in order to obtain credentials for authentication to the client. However, if one of the anonymous cipher suites is selected, then the server's `KeyManager` keystore is not necessary. And, unless the server requires client authentication, the peer acting as the client does not need a `KeyManager` keystore. Thus, in these situations it may be OK if no `javax.net.ssl.keyStore` system property value is defined.

Similarly, if a truststore is specified by the `javax.net.ssl.trustStore` system property, then the `TrustManager` created by the default `SSLContext` will be a `TrustManager` implementation for managing the specified truststore. In this case, if such a property exists but the file it specifies does not, then no truststore is used. If no `javax.net.ssl.trustStore` property exists, then a default truststore is searched for. If a truststore named *`java-home`*`/lib/security/jssecacerts` is found, it is used. If not, then a truststore named *`java-home`*`/lib/security/cacerts` is searched for and used (if it exists). Finally, if a truststore is still not found, then the truststore managed by the `TrustManager` will be a new empty truststore.

> **Note**
>
> The JDK ships with a limited number of trusted root certificates in the `java-home/lib/security/cacerts` file. As documented in `keytool` in *Java Development Kit Tool Specifications*, it is your responsibility to maintain (that is, add and remove) the certificates contained in this file if you use this file as a truststore.
>
> Depending on the certificate configuration of the servers that you contact, you may need to add additional root certificates. Obtain the needed specific root certificates from the appropriate vendor.

If the `javax.net.ssl.keyStoreType` and/or `javax.net.ssl.keyStorePassword` system properties are also specified, then they are treated as the default `KeyManager` keystore type and password, respectively. If no type is specified, then the default type is that returned by the `KeyStore.getDefaultType()` method, which is the value of the `keystore.type` Security Property, or "pkcs12" if no such Security Property is specified. If no keystore password is specified, then it is assumed to be a blank string "".

Similarly, if the `javax.net.ssl.trustStoreType` and/or `javax.net.ssl.trustStorePassword` system properties are also specified, then they are treated as the default truststore type and password, respectively. If no type is specified, then the default type is that returned by the `KeyStore.getDefaultType()` method. If no truststore password is specified, then it is assumed to be a blank string "".

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> This section describes the current JSSE reference implementation behavior. The system properties described in this section are not guaranteed to continue to have the same names and types (system or security) or even to exist at all in future releases. They are also not guaranteed to be examined and used by any other JSSE implementations. If they *are* examined by an implementation, then that implementation should handle them in the same manner as the JSSE reference implementation does, as described herein.

Customizing the Default Key Managers and Trust Managers

As noted in Customizing the Default Keystores and Truststores, Store Types, and Store Passwords, whenever a default `SSLSocketFactory` or `SSLServerSocketFactory` is created, and this default `SSLSocketFactory` (or `SSLServerSocketFactory`) comes from the JSSE reference implementation, a default `SSLContext` is associated with the socket factory.

This default `SSLContext` is initialized with a `KeyManager` and a `TrustManager`. The `KeyManager` and/or `TrustManager` supplied to the default `SSLContext` will be an implementation for managing the specified keystore or truststore, as described in the aforementioned section.

The `KeyManager` implementation chosen is determined by first examining the `ssl.KeyManagerFactory.algorithm` Security Property. If such a property value is specified, then a `KeyManagerFactory` implementation for the specified algorithm is searched for. The implementation from the first provider that supplies an implementation is used. Its `getKeyManagers()` method is called to determine the `KeyManager` to supply to the default `SSLContext`. Technically, `getKeyManagers()` returns an array of `KeyManager` objects, one `KeyManager` for each type of key material. If no such Security Property value is specified, then the default value of SunX509 is used to perform the search.

> **Note**
>
> A `KeyManagerFactory` implementation for the SunX509 algorithm is supplied by the SunJSSE provider. The `KeyManager` that it specifies is a `javax.net.ssl.X509KeyManager` implementation.

Similarly, the `TrustManager` implementation chosen is determined by first examining the `ssl.TrustManagerFactory.algorithm` Security Property. If such a property value is specified, then a `TrustManagerFactory` implementation for the specified algorithm is searched for. The implementation from the first provider that supplies an implementation is used. Its `getTrustManagers()` method is called to determine the `TrustManager` to supply to the default `SSLContext`. Technically, `getTrustManagers()` returns an array of `TrustManager` objects, one `TrustManager` for each type of trust material. If no such Security Property value is specified, then the default value of PKIX is used to perform the search.

> **Note**
>
> A `TrustManagerFactory` implementation for the PKIX algorithm is supplied by the SunJSSE provider. The `TrustManager` that it specifies is a `javax.net.ssl.X509TrustManager` implementation.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> This section describes the current JSSE reference implementation behavior. The system properties described in this section are not guaranteed to continue to have the same names and types (system or security) or even to exist at all in future releases. They are also not guaranteed to be examined and used by any other JSSE implementations. If they *are* examined by an implementation, then that implementation should handle them in the same manner as the JSSE reference implementation does, as described herein.

Disabled and Restricted Cryptographic Algorithms

In some environments, certain algorithms or key lengths may be undesirable when using TLS/DTLS. The Oracle JDK uses the `jdk.certpath.disabledAlgorithms`, `jdk.tls.disabledAlgorithm`, and `jdk.crypto.disabledAlgorithms` Security Properties to disable algorithms during TLS/DTLS protocol negotiation, including version negotiation, cipher suites selection, peer authentication, and key exchange mechanisms. Note that these Security Properties are not guaranteed to be used by other JDK implementations. See the *<java-home>*`/conf/security/java.security` file for information about the syntax of these Security Properties and their current active values.

- **`jdk.certpath.disabledAlgorithms` Property**: `CertPath` code uses the `jdk.certpath.disabledAlgorithms` Security Property to determine which algorithms should not be allowed during `CertPath` checking. For example, when a TLS server sends an identifying certificate chain, a client `TrustManager` that uses a `CertPath` implementation to verify the received chain will not allow the stated conditions. For example, the following line blocks any MD2-based certificate, as well as SHA1 TLSServer certificates that chain to trust anchors that are pre-installed in the `cacaerts` keystore. Likewise, this line blocks any RSA key less than 1024 bits.

  ```
  jdk.certpath.disabledAlgorithms=MD2, SHA1 jdkCA & usage TLSServer, RSA
  keySize < 1024
  ```

- **`jdk.tls.disabledAlgorithms` Property**: `SunJSSE` code uses the `jdk.tls.disabledAlgorithms` Security Property to disable TLS/DTLS protocols, cipher suites, keys, and so on. The syntax is similar to the `jdk.certpath.disabledAlgorithms` Security Property. For example, the following line disables the SSLv3 algorithm and all of the TLS_*_RC4_* cipher suites:

  ```
  jdk.tls.disabledAlgorithms=SSLv3, RC4
  ```

- **`jdk.crypto.disabledAlgorithms` Property**: In some environments, certain algorithms may be undesirable for certain cryptographic services. For example, MD2 is generally no longer considered to be a secure hash algorithm. This Security Property enables you to disable algorithms at the JCA/JCE level based on service name and algorithm name. For example, the following line disables the RSA/ECB/PKCS1Padding algorithm for the `Cipher` service and the MD2 algorithm for the `MessageDigest` service:

  ```
  jdk.crypto.disabledAlgorithms=Cipher.RSA/ECB/PKCS1Padding,
  MessageDigest.MD2
  ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

You can override the value of this Security Property by setting the value of the system property of the same name. This enables you to re-enable algorithms for applications that require them.

> **Note**
>
> The algorithm restrictions specified by these Security Properties do not apply to trust anchors or self-signed certificates.

If you require a particular condition, you can reactivate it by either removing the associated value in the Security Property in the `java.security` file or dynamically setting the proper Security Property before JSSE is initialized.

> **Note**
>
> Contact your security architect before modifying these Security Properties or enabling a cipher suite that hasn't been enabled; this allows the use of cipher suites with weaker protections.

Note that these Security Properties effectively create a third set of cipher suites, Disabled. The following list describes these three sets:

- **Disabled**: If a cipher suite contains any components (for example, RC4) on the disabled list (for example, `RC4` is specified in the `jdk.tls.disabledAlgorithms` Security Property), then that cipher suite is disabled and will **not** be considered for a connection handshake.
- **Enabled**: A list of specific cipher suites that will be considered for a connection.
- **Not Enabled**: A list of non-disabled cipher suites that will **not** be considered for a connection. To re-enable these cipher suites, call the appropriate `setEnabledCipherSuites()` or `setSSLParameters()` methods.

If any application attempts to reenable a cipher suite, which has been disabled by the `jdk.tls.disabledAlgorithms` Security Property, through the `setEnabledCipherSuites()` or `setSSLParameters()` methods, then JSSE allows the method call but does not allow the use of the disabled cipher suite during handshaking.

See SunJSSE Cipher Suites for a list of currently implemented SunJSSE cipher suites for this JDK release.

Legacy Cryptographic Algorithms

In some environments, a certain algorithm may be undesirable but it cannot be disabled because of its use in legacy applications. Legacy algorithms may still be supported, but applications should not use them as the security strength of legacy algorithms is usually not strong enough. During TLS/DTLS security parameters negotiation, legacy algorithms are not negotiated unless there are no other candidates. The Security Property `jdk.tls.legacyAlgorithms` specifies which algorithms the Oracle JDK considers as legacy algorithms. `<java-home>/conf/security/java.security` file for the syntax of this Security Property.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> - If a legacy algorithm is also restricted through the `jdk.tls.disabledAlgorithms` property or the `java.security.AlgorithmConstraints` API (see the method `javax.net.ssl.SSLParameters.setAlgorithmConstraints`), then the algorithm is completely disabled and will not be negotiated.
> - If your application uses an algorithm specified in the Security Property `jdk.tls.legacyAlgorithms`, use an alternative algorithm as soon as possible; a future JDK release may specify a legacy algorithm as a restricted algorithm.

Customizing the Encryption Algorithm Providers

The SunJSSE provider uses the SunJCE implementation for all its cryptographic needs. Although it is recommended that you leave the provider at its regular position, you can use implementations from other JCA or JCE providers by registering them *before* the SunJCE provider.

The standard JCA mechanism (see How Provider Implementations Are Requested and Supplied) can be used to configure providers, either statically via the security properties file `<java-home>/conf/security/java.security`, or dynamically via the `addProvider()` or `insertProviderAt()` method in the `java.security.Security` class.

Customizing the Size of Ephemeral Diffie-Hellman Keys

In TLS/DTLS connections, ephemeral Diffie-Hellman (DH) keys may be used internally during the handshaking. The SunJSSE provider provides a flexible approach to customize the strength of the ephemeral DH key size during TLS/DTLS handshaking.

Diffie-Hellman (DH) keys of sizes less than 2048 bits have been deprecated because of their insufficient strength. You can customize the ephemeral DH key size with the system property `jdk.tls.ephemeralDHKeySize`. This system property does not impact DH key sizes in `ServerKeyExchange` messages for exportable cipher suites. It impacts only the DHE_RSA, DHE_DSS, and DH_anon-based cipher suites in the JSSE Oracle provider.

> **Note**
>
> Unless the `jdk.tls.ephemeralDHKeySize` system property is set to `legacy`, the SunJSSE implementation will first try to negotiate a common DH group using FFDHE, which is a TLS extension defined by RFC 7919. If the SunJSSE implementation can negotiate a group, then it will use the size defined by that group. Otherwise, it will fallback to using a keysize as described in this section. FFDHE is enabled by default, but you can disable it by setting the system property `jsse.enableFFDHE` to `false`.

You can specify one of the following values for this property:

- Undefined: A DH key of size 2048 bits will be used always for non-exportable cipher suites. This is the default value for this property.
- `legacy`: The JSSE Oracle provider preserves the legacy behavior (for example, using ephemeral DH keys of sizes 512 bits and 768 bits) of JDK 7 and earlier releases.
- `matched`:

Copyright © 1993, 2026, Oracle and/or its affiliates.

  - For non-exportable anonymous cipher suites, the DH key size in ServerKeyExchange messages is 2048 bits.
  - For X.509 certificate-based authentication (of non-exportable cipher suites), the DH key size matching the corresponding authentication key is used, except that a fixed size of 1024 bits is used for any key smaller than 1024 bits, and a fixed size of 2048 bits is used for any key larger than 2048 bits.

    For example, if the public key size of an authentication certificate is 2048 bits, then the ephemeral DH key size is 2048 bits unless the cipher suite is exportable. This key sizing scheme keeps the cryptographic strength consistent between authentication keys and key-exchange keys.

- A valid integer between 1024 and 8192 in multiples of 64, inclusively: A fixed ephemeral DH key size of the specified value, in bits, will be used for non-exportable cipher suites.

The following table summarizes the minimum and maximum acceptable DH key sizes for each of the possible values for the system property `jdk.tls.ephemeralDHKeySize`:

Table 8-4 DH Key Sizes for the System Property `jdk.tls.ephemeralDHKeySize`

Value of jdk.tls.ephemeralDHKeySize	Undefined	legacy	matched	Integer value (fixed)
Exportable DH key size	512	512	512	512
Non-exportable anonymous cipher suites	2048	768	2048	A valid integer between 1024 and 8192 in multiples of 64, inclusively: A fixed ephemeral DH key size of the specified value, in bits, will be used for non-exportable cipher suites.
Authentication certificate	2048	768	The key size is the same as the authentication certificate unless the key is less than 1024 bits or greater than 2048 bits. If the key is less than 1024 bits, then a DH key of 1024 bits is used. If the key is greater than 2048 bits, then a DH key of 2048 bits is used. Consequently, you may use the values 1024 or 2048 only.	The fixed key size is specified by a valid integer property value, which must be between 1024 and 8192 in multiples of 64, inclusively.

Customizing the Maximum Fragment Length Negotiation (MFLN) Extension

In order to negotiate smaller maximum fragment lengths, clients have an option to include an extension of type `max_fragment_length` in the ClientHello message. A system property

Copyright © 1993, 2026, Oracle and/or its affiliates.

`jsse.enableMFLNExtension` can be used to enable or disable the MFLN extension for TLS/DTLS.

Maximum Fragment Length Negotiation

It may be desirable for constrained TLS/DTLS clients to negotiate a smaller maximum fragment length due to memory limitations or bandwidth limitations. In order to negotiate smaller maximum fragment lengths, clients have an option to include an extension of type `max_fragment_length` in the (extended) ClientHello message. See RFC 6066.

Once a maximum fragment length has been successfully negotiated, the TLS/DTLS client and server can immediately begin fragmenting messages (including handshake messages) to ensure that no fragment larger than the negotiated length is sent.

System Property jsse.enableMFLNExtension

A system property `jsse.enableMFLNExtension` is defined to enable or disable the MFLN extension. The `jsse.enableMFLNExtension` is disabled by default.

The value of the system property can be set as follows:

Table 8-5 jsse.enableMFLNExtension system property

System Property	Description
`jsse.enableMFLNExtension=true`	Enable the MFLN extension. If the returned value of `SSLParameters.getMaximumPacketSize()` is less than (2^{12} + header-size) the maximum fragment length negotiation extension would be enabled.
`jsse.enableMFLNExtension=false`	Disable the MFLN extension.

Configuring the Maximum and Minimum Packet Size

Set the maximum expected network packet size in bytes for a TLS/DTLS record with the `SSLParameters.setMaximumPacketSize` method.

It is recommended that the packet size should not be less than 256 bytes so that small handshake messages, such as HelloVerifyRequests, are not fragmented.

Limiting Amount of Data Algorithms May Encrypt with a Set of Keys

You can specify a limit on the amount of data an algorithm may encrypt with a specific set of keys with the `jdk.tls.keyLimits` Security Property. Once this limit is reached, a KeyUpdate post-handshake message is sent, which requests that the current set of keys be updated. This Security Property is only for symmetrical ciphers with TLS 1.3.

The syntax for this property is as follows:

```
jdk.tls.keyLimits=KeyLimit { , KeyLimit }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

KeyLimit

AlgorithmName KeyUpdate *Length*

AlgorithmName
A full algorithm transformation

Length
The amount of encrypted data in a session before a KeyUpdate message is sent. This value may be an integer value in bytes or as a power of two, for example, 2^37.

For example, the following specifies that a KeyUpdate message is sent once the algorithm AES/GCM/NoPadding has encrypted 2^{37} bytes:

```
jdk.tls.keyLimits=AES/GCM/NoPadding KeyUpdate 2^37
```

Resuming Session Without Server-Side State

The feature session resumption without server-site state enables the server side of JSSE to operate stateless. As described in RFC 5077 (TLS Session Resumption Without Server-Side State) for TLS 1.2 and earlier and RFC 8446 for TLS 1.3, the TLS server sends internal session information in the form of an encrypted session ticket to a client that supports stateless operation. That session ticket is presented to the server during the TLS handshake to resume the session. This should improve the performance and memory usage of the TLS server under large workloads as the session cache will seldom be used. However, with less session information cached, some session information may not be available. This feature is enabled by default; you can turn it off by setting two system properties:

- `jdk.tls.client.enableSessionTicketExtension`: Toggles the session ticket extension in the ClientHello message on the client side for TLS 1.2 and earlier. A value of `true` (default value) sends the extension, `false` does not.
- `jdk.tls.server.enableSessionTicketExtension`: Enables a server to use stateless session tickets if the client supports it. Clients that don't support stateless session tickets use the cache instead. A value of `true` (default value) enables the use of stateless session tickets and stateless session resumption: the NewSessionTicket message includes all session information (in an encrypted format). A value of `false` disables the use of stateless session tickets, which means that session resumption is stateful: the NewSessionTicket message just contains a key that is used by the server during session resumption to access the session information from its session cache.

> **Note**
>
> For TLS 1.3, stateless tickets use the existing pre-shared key (PSK) resumption extension (see Session Resumption with a Pre-Shared Key). Therefore, session resumption without server-site state doesn't require these two properties. However, the contents of stateless tickets, in particular, the contents of a NewSessionTicket message, depend on the value of `jdk.tls.server.enableSessionTicketExtension`: If `jdk.tls.server.enableSessionTicketExtension` is `true`, then the NewSessionTicket message contains the encrypted session state. If `false`, then the session state is cached with the PSK resumption extension.
>
> For TLS 1.2, stateless session tickets are used only if they are supported by the client.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Specifying That close_notify Alert Is Sent When One Is Received

If the `jdk.tls.acknowledgeCloseNotify` system property is set to `true`, then when the client or server receives a close_notify alert, it sends a corresponding close_notify alert and the connection is duplex closed.

TLS 1.2 and earlier versions use a duplex-close policy. However, TLS 1.3 uses a half-close policy, which means that the inbound and the outbound close_notify alerts are independent. When upgrading to TLS 1.3, unexpected behavior can occur if your application shuts down the TLS/DTLS connection by using only one of the `SSLEngine.closeInbound()` or `SSLEngine.closeOutbound()` methods but not both on each side of the connection. If your application unexpectedly hangs or times out when the underlying TLS/DTLS transportation is not duplex closed, you may need to set this property to `true`.

Note that when a TLS/DTLS connection is no longer needed, the client and server applications should each close both sides of their respective connection.

Enabling certificate_authorities Extension for Server Certificate Selection

The certificate_authorities extension is an optional extension introduced in TLS 1.3. It's used to indicate the certificate authorities (CAs) that an endpoint supports and should be used by the receiving endpoint to guide certificate selection.

This extension is always present for client certificate selection, while it's optional for server certificate selection.

Enable this extension for server certificate selection by setting the system property `jdk.tls.client.enableCAExtension` to `true`. The default value of the property is `false`.

> **Note**
>
> If the client or server trusts more CAs such that it exceeds the size limit of the extension (less than 2^16 bytes), then the extension is not enabled. Also, some server implementations don't allow handshake messages to exceed 2^14 bytes. Thus, there may be interoperability issues if `jdk.tls.client.enableCAExtension` is set to `true` and the client trusts more CAs such that it exceeds the server implementation limit.

SunJSSE Renegotiation Interoperability Modes

The SunJSSE implementation reenables renegotiations by default for connections to peers compliant with RFC 5746. That is, both the client and server **must support RFC 5746** to securely renegotiate. (RFC 5746 addresses a flaw that was discovered in SSL/TLS protocols.) SunJSSE provides some interoperability modes for connections with peers that have not been upgraded, but users are **strongly encouraged to update both their client and server implementations as soon as possible**.

SunJSSE has three renegotiation interoperability modes. Each mode fully supports RFC 5746: Transport Layer Security (TLS) Renegotiation Indication Extension, but has these added semantics when communicating with a peer that has not been upgraded:

- **Strict mode**: Requires both client and server be upgraded to RFC 5746 and to send the proper RFC 5746 messages. If not, the initial (or subsequent) handshaking will fail and the connection will be terminated.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- **Interoperable mode (default)**: Use of the proper RFC 5746 messages is optional; however, legacy (original SSL/TLS specifications) renegotiations are disabled if the proper messages are not used. Initial legacy connections are still allowed, but legacy renegotiations are disabled. This is the best mix of security and interoperability, and is the default setting.
- **Insecure mode**: Permits full legacy renegotiation. Most interoperable with legacy peers but vulnerable to the original MITM attack.

The three mode distinctions only affect a connection with a peer that has not been upgraded. Ideally, strict (full RFC 5746) mode should be used for all clients and servers; however, it will take some time for all deployed SSL/TLS implementations to support RFC 5746, because the interoperable mode is the current default.

The following table contains interoperability information about the modes for various cases in which the client and/or server are either updated to support RFC 5746 or not.

Table 8-6 Interoperability Information

Client	Server	Mode
Updated	Updated	Secure renegotiation in all modes.
Legacy[1]	Updated	• **Strict** If clients do not send the proper RFC 5746 messages, then initial connections will immediately be terminated by the server (`SSLHandshakeException` or `handshake_failure`). • **Interoperable** Initial connections from legacy clients are allowed (missing RFC 5746 messages), but renegotiations will not be allowed by the server. [2][3] • **Insecure** Connections and renegotiations with legacy clients are allowed, but are vulnerable to the original MITM attack.
Updated	Legacy [1]	• **Strict** If the server does not respond with the proper RFC 5746 messages, then the client will immediately terminate the connection (`SSLHandshakeException` or `handshake_failure`). • **Interoperable** Initial connections from legacy servers are allowed (missing RFC 5746 messages), but renegotiations will not be allowed by the server. [2][3] • **Insecure** Connections and renegotiations with legacy servers are allowed, but are vulnerable to the original MITM attack.
Legacy [1]	Legacy [1]	Existing SSL/TLS behavior, vulnerable to the MITM attack.

1 "Legacy" means the original SSL/TLS specifications (that is, *not* RFC 5746).

2 If renegotiations are reenabled, then they will be treated as "Legacy" by the peer that is compliant with RFC 5746, because they do not send the proper RFC 5746 messages.

3 In SSL/TLS, renegotiations can be initiated by either side. Applications communicating with a peer that has not been upgraded in Interoperable mode and that attempt to initiate renegotiation (via `SSLSocket.startHandshake()`

Copyright © 1993, 2026, Oracle and/or its affiliates.

or `SSLEngine.beginHandshake()`) will receive an `SSLHandshakeException` (`IOException`) and the connection will be shut down (`handshake_failure`). Applications that receive a renegotiation request from a peer that has not been upgraded will respond according to the type of connection in place:

- **TLSv1** A warning alert message of type `no_renegotiation(100)` will be sent to the peer and the connection will remain open. Older versions of SunJSSE will shut down the connection when a `no_renegotiation` alert is received.
- **SSLv3** The application will receive an `SSLHandshakeException`, and the connection will be closed (`handshake_failure`). The `no_renegotiation` alert is not defined in the SSLv3 specification.

Set the mode with the the following system properties (see How to Specify a java.lang.System Property):

- `sun.security.ssl.allowUnsafeRenegotiation` controls whether legacy (unsafe) renegotiations are permitted.
- `sun.security.ssl.allowLegacyHelloMessages` allows the peer to perform the handshake process without requiring the proper RFC 5746 messages.

> **Note**
>
> The system properties `sun.security.ssl.allowUnsafeRenegotiation` and `sun.security.ssl.allowLegacyHelloMessages` are deprecated and might be removed in a future JDK release.

Table 8-7 Values of the System Properties for Setting the Interoperability Mode

Mode	allowLegacyHelloMessages	allowUnsafeRenegotiation
Strict	`false`	`false`
Interoperable (default)	`true`	`false`
Insecure	`true`	`true`

> **Caution**
>
> Do not reenable the insecure SSL/TLS renegotiation, as this would reestablish the vulnerability that was discovered in SSL/TLS protocols.

Workarounds and Alternatives to SSL/TLS Renegotiation

All peers should be updated to RFC 5746-compliant implementation as soon as possible. Even with this RFC 5746 fix, communications with peers that have not been upgraded will be affected if a renegotiation is necessary. Here are a few suggested options:

- **Restructure the peer to not require renegotiation.**

  Renegotiations are typically used by web servers that initially allow for anonymous client browsing but later require SSL/TLS authenticated clients, or that may initially allow weak cipher suites but later need stronger ones. The alternative is to require client authentication or strong cipher suites during the *initial* negotiation. There are a couple of options for doing so:

  - If an application has a browse mode until a certain point is reached and a renegotiation is required, then you can restructure the server to eliminate the browse mode and require all initial connections be strong.

Copyright © 1993, 2026, Oracle and/or its affiliates.

  - Break the server into two entities, with the browse mode occurring on one entity, and using a second entity for the more secure mode. When the renegotiation point is reached, transfer any relevant information between the servers.

  Both of these options require a fair amount of work, but will not reopen the original security flaw.

- **Set renegotiation interoperability mode to "insecure" using the system properties.**

  See SunJSSE Renegotiation Interoperability Modes.

Allow Unsafe Server Certificate Change in SSL/TLS Renegotiations

Server certificate change in an SSL/TLS renegotiation may be unsafe if all of the following apply:

- Endpoint identification is not enabled in an SSL/TLS handshaking.
- The previous handshake is a session-resumption abbreviated initial handshake.
- The identities represented by both certificates can be regarded as different. The following steps are followed to determine if two certificates represent the same identity:
  1. If the subject alternative names of IP address are present in both certificates, then they are identical.
  2. Otherwise, if the subject alternative names of DNS name are present in both certificates, then they are identical.
  3. Otherwise, if the subject fields are present in both certificates, then the certificate subjects and issuers are identical.

Unsafe server certificate change in SSL/TLS renegotiations is not allowed by default. Use the system property `jdk.tls.allowUnsafeServerCertChange` to define whether unsafe server certificate change in an SSL/TLS renegotiation should be restricted or not. The default value of this system property is `false`.

> ⚠ **Caution**
>
> Do not set the system property to `true` unless it is really necessary, as this would re-establish the unsafe server certificate change vulnerability.

Client-Driven OCSP and OCSP Stapling

Use the Online Certificate Status Protocol (OCSP) to determine the X.509 certificate revocation status during the Transport Layer Security (TLS) handshake.

X.509 certificates used in TLS can be revoked by the issuing Certificate Authority (CA) if there is reason to believe that a certificate is compromised. You can check the revocation status of certificates during the TLS handshake by using one of the following approaches.

- **Certificate Revocation List (CRL)**: A CRL is a simple list of revoked certificates. The application receiving a certificate gets the CRL from a CRL server and checks if the certificate received is on the list. There are two disadvantages to using CRLs that mean a certificate could be revoked:
  - CRLs can become very large so there can be a substantial increase in network traffic.

Copyright © 1993, 2026, Oracle and/or its affiliates.

  - Many CRLs are created with longer validity periods, which increases the possibility of a certificate being revoked within that validity period and not showing up until the next CRL refresh.

  See Certificate/CRL Storage Classes in Java PKI Programmer's Guide.

- **Client-driven OCSP** : In client-driven OCSP, the client uses OCSP to contact an OCSP responder to check the certificate's revocation status. The amount of data required is usually less than that of a CRL, and the OCSP responder is likely to be more up-to-date with the revocation status than a CRL. Each client connecting to a server requires an OCSP response for each certificate being checked. If the server is a popular one, and many of the clients are using clientdriven OCSP, these OCSP requests can have a negative effect on the performance of the OCSP responder.
- **OCSP stapling**: OCSP stapling enables the server, rather than the client, to make the request to the OCSP responder. The server staples the OCSP response to the certificate and returns it to the client during the TLS handshake. This approach enables the presenter of the certificate, rather than the issuing CA, to bear the resource cost of providing OCSP responses. It also enables the server to cache the OCSP responses and supply them to all clients. This significantly reduces the load on the OCSP responder because the response can be cached and periodically refreshed by the server rather than by each client.

Client-Driven OCSP and Certificate Revocation

Client-driven Online Certificate Status Protocol (OCSP) enables the client to check the certificate revocation status by connecting to an OCSP responder during the Transport Layer Security (TLS) handshake.

The client-driven OCSP request occurs during the TLS handshake just after the client receives the certificate from the server and validates it.

TLS Handshake with Client-Driven OCSP

Client-driven OCSP is used during the TLS handshake between the client and the server to check the server certificate revocation status. After the client receives the certificate, it performs certificate validation. If the validation is successful, then the client verifies that the certificate was not revoked by the issuer. This is done by sending an OCSP request to an OCSP responder. After receiving the OCSP response, the client checks this response before completing the TLS handshake.

Usually the client finds the OCSP responder's URL by looking in the Authority Information Access (AIA) extension of the certificate, but it can be set to a static URL through the use of a system property.

Setting up a Java Client to use Client-Driven OCSP

Client-driven OCSP is enabled by enabling revocation checking and enabling OCSP.

To configure a Java client to use client-driven OCSP, the Java client must already be set up to connect to a server using TLS.

1. Enable revocation checking. You can do this in two different ways.
   - Set the system property `com.sun.net.ssl.checkRevocation` to `true`.
   - Use the `setRevocationEnabled` method on `PKIXParameters`. See The PKIXParameters Class.
2. Enable client-driven OCSP:

   Set the Security Property `ocsp.enable` to `true`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Both steps are necessary. The `ocsp.enable` setting has no effect unless revocation checking is enabled.

OCSP Stapling and Certificate Revocation

Online Certificate Status Protocol (OCSP) stapling enables the presenter of a certificate, rather than the issuing Certificate Authority (CA), to bear the resource cost of providing the OCSP responses that contain the certificate's revocation status.

TLS Handshake with OCSP Stapling

OCSP stapling is used during the Transport Layer Security (TLS) handshake between the client and the server to check the server certificate revocation status. The server makes the OCSP request to the OCSP responder and staples the OCSP responses to the certificates returned to the client. By having the server make the request to the OCSP responder, the responses can be cached, and then used multiple times for many clients.

The client receiving the certificates with stapled OCSP responses validates each certificate, and then checks the OCSP responses before continuing with the handshake. If, from the client's perspective, the stapled OCSP response from the server for a certificate is missing, the client will attempt to use client-driven OCSP or Certificate Revocation Lists (CRLs) to get revocation information if the following are true:

- The RevocationEnabled flag is set to `true` through the `PKIXParameters.setRecovcationEnabled` method.
- OCSP checking is enabled by setting the `ocsp.enable` Security Property to `true`.

OCSP checking works in conjunction with CRLs during revocation checking. See Appendix C: OCSP Support in Java PKI Programmer's Guide.

Status Request Versus Multiple Status Request

The OCSP stapling feature implements the TLS Certificate Status Request extension (section 8 of RFC 6066) and the Multiple Certificate Status Request Extension (RFC 6961).

The TLS Certificate Status Request extension requests revocation information for only the server certificate in the certificate chain while the Multiple Certificate Status Request Extension requests revocation information for all certificates in the certificate chain. In the case where only the server certificate's revocation information is sent to the client, other certificates in the chain may be verified using the Certificate Revocation Lists (CRLs) or client-driven OCSP (but the client will need to be set up to do this).

Although TLS allows the server to also request the client's certificate, there is no provision in OCSP stapling that enables the client to contact the appropriate OCSP responder and staple the response to the certificate sent to the server.

The OCSP Request and Response

OCSP request and response messages are usually sent over unencrypted HTTP. The response is signed by the CA.

If necessary, the stapled responses can be obtained in the client code by calling the `getStatusResponses` method on the `ExtendedSSLSession` object. The method signature is:

```
public List<byte[]> getStatusResponses();
```

The OCSP response is encoded using the Distinguished Encoding Rules (DER) in a format described by the ASN.1 found in RFC 6960.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Setting Up a Java Client to Use OCSP Stapling

Online Certificate Status Protocol (OCSP) stapling is enabled on the client side by setting the system property `jdk.tls.client.enableStatusRequestExtension` to `true` (its default value).

To configure a Java client to make use of the OCSP response stapled to the certificate returned by a server, the Java client must already be set up to connect to a server using TLS, and the server must be set up to staple an OCSP response to the certificate it returns part of the TLS handshake.

1. Enable OCSP stapling on the client:

   If necessary, set the system property `jdk.tls.client.enableStatusRequestExtension` to `true`.

2. Enable revocation checking. You can do this in two different ways.

   - Set the system property `com.sun.net.ssl.checkRevocation` to `true`. You can do this from the command line or in the code.
   - Use the `setRevocationEnabled` method on the `PKIXParameters` class. See The PKIXParameters Class.

   For the client to include the stapled responses received from the server in the certificate validation, revocation checking must be set to `true`. If revocation checking is not set to `true`, then the connection will be allowed to proceed regardless of the presence or status of the revocation information

Setting Up a Java Server to Use OCSP Stapling

Online Certificate Status Protocol (OCSP) stapling is enabled on the server by setting the system property `jdk.tls.server.enableStatusRequestExtension` to `true`. (It is set to `false` by default.)

The following steps can be used to configure a Java server to connect to an OCSP responder and staple the OCSP response to the certificate to be returned to the client. The Java server must already be set up to respond to clients using TLS.

1. Enable OCSP stapling on the server:

   Set the system property `jdk.tls.server.enableStatusRequestExtension` to `true`.

2. Optional: Set other properties as required. See OCSP Stapling Configuration Properties for a list of the valid properties.

OCSP Stapling Configuration Properties

This topic lists the effects of setting various properties when using the Online Certificate Status Protocol (OCSP). It shows the properties used in both client-driven OCSP and OCSP stapling.

Server-side Properties

Most of the properties are read at `SSLContext` instantiation time. This means that if you set a property, you must obtain a new `SSLContext` object so that an `SSLSocket` or `SSLEngine` object you obtain from that `SSLContext` object will reflect the property setting. The one exception is the `jdk.tls.stapling.responseTimeout` property. That property is evaluated when the `ServerHandshaker` object is created (essentially at the same time that an `SSLSocket` or `SSLEngine` object gets created).

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-8 Server-Side OCSP stapling Properties

Property	Description	Default Value
`jdk.tls.server.enableStatusRequestExtension`	Enables the server-side support for OCSP stapling.	False
`jdk.tls.stapling.responseTimeout`	Controls the maximum amount of time the server will use to obtain OCSP responses, whether from the cache or by contacting an OCSP responder. The responses that are already received will be sent in a `CertificateStatus` message, if applicable based on the type of stapling being done.	5000 (integer value in milliseconds)
`jdk.tls.stapling.cacheSize`	Controls the maximum cache size in entries. If the cache is full and a new response needs to be cached, then the least recently used cache entry will be replaced with the new one. A value of zero or less for this property means that the cache will have no upper bound on the number of responses it can contain.	256 objects
`jdk.tls.stapling.cacheLifetime`	Controls the maximum life of a cached response. It is possible for responses to have shorter lifetimes than the value set with this property if the response has a **nextUpdate** field that expires sooner than the cache lifetime. A value of zero or less for this property disables the cache lifetime. If an object has no **nextUpdate** value and cache lifetimes are disabled, then the response will not be cached.	3600 seconds (1 hour)
`jdk.tls.stapling.responderURI`	Enables the administrator to set a default URI in the event that certificates used for TLS do not have the Authority Info Access (AIA) extension. It will not override the Authority Info Access extension value unless the `jdk.tls.stapling.responderOverride` property is set.	Not set
`jdk.tls.stapling.responderOverride`	Enables a URI provided through the `jdk.tls.stapling.responderURI` property to override any AIA extension value.	False
`jdk.tls.stapling.ignoreExtensions`	Disables the forwarding of OCSP extensions specified in the `status_request` or `status_request_v2` TLS extensions.	False

Client-Side Settings

Table 8-9 Client-Side Settings Used in OCSP Stapling

PKIXBuilderParameters	checkRevocation Property	PKIXRevocationChecker	Result
Default	Default	Default	Revocation checking is disabled.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-9 (Cont.) Client-Side Settings Used in OCSP Stapling

PKIXBuilderParameters	checkRevocationProperty	PKIXRevocationChecker	Result
Default	True	Default	Revocation checking is enabled.[1]
Instantiated	Default	Default	Revocation checking is enabled.[1]
Instantiated	Default	Instantiated, added to `PKIXBuilderParameters` object.	Revocation checking is enabled and[1]will behave according to the `PKIXRevocationChecker` settings.

Footnote 1 Note that client-side OCSP fallback will occur only if the `ocsp.enable` Security Property is set to `true`.

Developers have some flexibility in how to handle the responses provided through OCSP stapling. OCSP stapling makes no changes to the current methodologies involved in certificate path checking and revocation checking. This means that it is possible to have both client and server assert the `status_request` extensions, obtain OCSP responses through the `CertificateStatus` message, and provide user flexibility in how to react to revocation information, or the lack thereof.

If no `PKIXBuilderParameters` is provided by the caller, then revocation checking is disabled. If the caller creates a `PKIXBuilderParameters` object and uses the `setRevocationEnabled` method to enable revocation checking, then stapled OCSP responses will be evaluated. This is also the case if the `com.sun.net.ssl.checkRevocation` property is set to `true`.

Configuring Default Extensions

Some TLS implementations may not handle unknown extensions properly. As a result, you might encounter unexpected interoperability issues when the JDK introduces new extensions. Two system properties enable you to customize default extensions:

- `jdk.tls.client.disableExtensions`: Blocks extensions used on the client side.
- `jdk.tls.server.disableExtensions`: Blocks extensions used on the server side.

If an extension is disabled, then it won't be produced nor processed in handshake messages.

The value of these system properties is a list of comma-separated standard TLS extension names. See Transport Layer Security (TLS) Extensions for a list of these names. Extension names are case-sensitive, and unknown, unsupported misspelled and duplicated names are ignored.

> **Note**
>
> Although system properties exist that enable and disable specific TLS extensions, such as `jsse.enableMFLNExtension`, `jsse.enableSNIExtension`, and `jsse.enableSNIExtension`, an extension won't be enabled if it's disabled through `jdk.tls.client.disableExtensions` or `jdk.tls.server.disableExtensions`, even though it could be enabled though the corresponding system property.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Configuring TLS 1.3 Certificate Compression

By default, the JDK supports TLS 1.3 certificate compression with the `zlib` compression algorithm. This feature can reduce the size of certificate chains exchanged during TLS handshakes.

To disable it, add the `compress_certificate` extension to the `jdk.tls.client.disableExtensions` and `jdk.tls.server.disableExtensions` system properties. For example, the following Java command-line options disable TLS certificate compression for both the client and the server:

```
-Djdk.tls.client.disableExtensions=compress_certificate
-Djdk.tls.server.disableExtensions=compress_certificate
```

See RFC 8879: TLS Certificate Compression for more information. Note that the `brotli` and `zstd` compression algorithms are not supported.

Specifying Signature Schemes That Can Be Used over the TLS/DTLS Protocols

Signature schemes are the algorithms used in the digital signatures of TLS/DTLS connections. The system property `jdk.tls.server.SignatureSchemes` contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS/DTLS connections on the server side. The `jdk.tls.client.SignatureSchemes` contains the same except that it's for connections on the client side.

For a list of signature scheme names, see the section Signature Schemes in *Java Security Standard Algorithm Names Specification*.

An application can set the signature schemes with the method `javax.net.ssl.SSLParameters.setSignatureSchemes(String[])`. See the method `javax.net.ssl.SSLParameters.getSignatureSchemes()` for specific details on how signature schemes are used in SSL/TLS/DTLS connections.

Customizing the Supported Named Groups for TLS/DTLS Key Exchange

Named groups are predefined cryptographic parameters for key exchange. The `jdk.tls.namedGroups` system property contains a comma-separated list within quotation marks of enabled named groups in preference order.

The standard list of key exchange named groups are defined in the Named Groups section of *Java Security Standard Algorithm Names Specification*.

Call `javax.net.ssl.SSLParameters.getNamedGroups()` to get a prioritized array of key exchange named groups names that can be used over the TLS/DTLS protocols. You can call `javax.net.ssl.SSLParameters.setNamedGroups(String[])` to specify a prioritized array of key exchange named groups names.

Enabling Hybrid Key Exchange Schemes

The JDK's TLS 1.3 implementation supports hybrid key exchange schemes that use a quantum-resistant Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM) algorithm as one component.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Hybrid key exchange involves using two (or more) key exchange algorithms based on different cryptographic assumptions to establish a shared secret. This shared secret is secure as long as one of the algorithms remains unbroken. The JDK's TLS 1.3 implementation supports key exchange schemes that combine ML-KEM with an Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) algorithm. See the section Named Groups in *Java Security Standard Algorithm Names* for a list of these supported key exchange schemes.

ML-KEM is a quantum-resistant algorithm designed for key encapsulation. See FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard for more information.

ECDHE is considered to be traditional, which is one that's currently widely deployed but may be deprecated in the future. You organization might mandate the use of traditional algorithms due to regulatory constraints from, for example, US National Institute of Standards and Technology (NIST) Federal Information Processing Standards (FIPS) compliance.

Hybrid key exchange enables you to become an early adopter of quantum-resistant algorithms while retaining the security offered by traditional algorithms. In addition, it enables potential security against retroactive decryption. This means that you can protect your users from adversaries recording TLS sessions and decrypting them later upon the arrival of a quantum computer or some other cryptographic breakthrough.

Using Hybrid Named Groups

By default, the most preferred named group is the hybrid key exchange scheme X25519MLKEM768. This means that by default users will negotiate TLS connections with a hybrid named group if the peer supports it as well.

You can change which named groups are supported and their preference order with the `jdk.tls.namedGroups` system property. Alternatively, you can dynamically specify the list of supported named groups in your application. The following example instructs the TLS client to prefer two traditional key exchange schemes, secp256r1 and x25519, over the default hybrid key exchange scheme X25519MLKEM768. It also enables an additional hybrid key exchange scheme, SecP256r1MLKEM768, with the method `SSLParameters.setNamedGroups(String[])`:

```
SSLSocket tlsSock = (SSLSocket)(SSLContext.getDefault().
    getSocketFactory().createSocket());
SSLParameters params = tlsSock.getSSLParameters();

// Configure the socket to use two traditional schemes
// and two hybrid KEM schemes
params.setNamedGroups(new String[] {
    "secp256r1", "x25519",
    "X25519MLKEM768", "SecP256r1MLKEM768"
});
tlsSock.setSSLParameters(params);
```

Setting the Maximum Certificate Chain Length Accepted from the Client or Server During TLS/DTLS Handshaking

You can set the maximum allowed length of the certificate chain accepted from the client or server during TLS/DTLS handshaking with the system properties `jdk.tls.server.maxInboundCertificateChainLength` and `jdk.tls.client.maxInboundCertificateChainLength`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

A service can function as both a TLS/DTLS server and client. When the service acts as a server, it enforces a maximum certificate chain length accepted from clients. When the service acts as a client, it enforces a maximum certificate chain length accepted from servers.

If you set a value for the system properties `jdk.tls.server.maxInboundCertificateChainLength` and `jdk.tls.client.maxInboundCertificateChainLength`, then they override the existing system property `jdk.tls.maxCertificateChainLength` as follows:

- If you set the system property `jdk.tls.server.maxInboundCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum length of a client certificate chain accepted by a server. Otherwise, if you have set the system property `jdk.tls.maxCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum client certificate chain length. If you haven't set either property, then the default value 8 is the maximum client certificate chain length.
- If you set the system property `jdk.tls.client.maxInboundCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum length of a server certificate chain accepted by a client. Otherwise, if you have set the system property `jdk.tls.maxCertificateChainLength` to a value equal to or greater than 0, then this value is the maximum server certificate chain length. If you haven't set either property, then the default value 10 is the maximum server certificate chain length.

Support for the TLS Session Hash and Extended Master Secret Extension

By default, the TLS session hash and extended master secret extension (RFC 7627) is supported. Note that in general, a server certificate change is restricted if endpoint identification is not enabled and the previous handshake is a session-resumption abbreviated initial handshake, unless the identities represented by both certificates can be regarded as the same. However, if the extension is enabled or negotiated, the server certificate changing restriction is not necessary and will be discarded accordingly.

In case of compatibility issues, an application may disable negotiation of this extension by setting the system property `jdk.tls.useExtendedMasterSecret` to `false`. By setting the system property `jdk.tls.allowLegacyResumption` to `false`, an application can reject abbreviated handshaking when the session hash and extended master secret extension are not negotiated. By setting the system property `jdk.tls.allowLegacyMasterSecret` to `false`, an application can reject connections that do not support the session hash and extended master secret extension.

Hardware Acceleration and Smartcard Support

The Java Cryptography Architecture (JCA) is a set of packages that provides a framework and implementations for encryption, key generation and key agreement, and message authentication code (MAC) algorithms. (See Java Cryptography Architecture (JCA) Reference Guide.) The SunJSSE provider uses JCA exclusively for all of its cryptographic operations and can automatically take advantage of JCE features and enhancements, including JCA's support for RSA PKCS#11. This support enables the SunJSSE provider to use hardware cryptographic accelerators for significant performance improvements and to use smartcards as keystores for greater flexibility in key and trust management. .

Use of hardware cryptographic accelerators is automatic if JCA has been configured to use the Oracle PKCS#11 provider, which in turn has been configured to use the underlying accelerator hardware. The provider must be configured before any other JCA providers in the provider list. For details on how to configure the Oracle PKCS#11 provider, see PKCS#11 Reference Guide.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Configuring JSSE to Use Smartcards as Keystores and Truststores

Support for PKCS#11 in JCA also enables access to smartcards as a keystore. For details on how to configure the type and location of the keystores to be used by JSSE, see Customizing JSSE. To use a smartcard as a keystore or truststore, set the `javax.net.ssl.keyStoreType` and `javax.net.ssl.trustStoreType` system properties, respectively, to `pkcs11`, and set the `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` system properties, respectively, to `NONE`. To specify the use of a specific provider, use the `javax.net.ssl.keyStoreProvider` and `javax.net.ssl.trustStoreProvider` system properties (for example, set them to `SunPKCS11-joe`). By using these properties, you can configure an application that previously depended on these properties to access a file-based keystore to use a smartcard keystore with no changes to the application.

Some applications request the use of keystores programmatically. These applications can continue to use the existing APIs to instantiate a `Keystore` and pass it to its key manager and trust manager. If the `Keystore` instance refers to a PKCS#11 keystore backed by a Smartcard, then the JSSE application will have access to the keys on the smartcard.

Multiple and Dynamic Keystores

Smartcards (and other removable tokens) have additional requirements for an `X509KeyManager`. Different smartcards can be present in a smartcard reader during the lifetime of a Java application, and they can be protected using different passwords.

The `KeyStore.Builder` class abstracts the construction and initialization of a `KeyStore` object. It supports the use of `CallbackHandler` for password prompting, and its subclasses can be used to support additional features as desired by an application. For example, it is possible to implement a `Builder` that allows individual `KeyStore` entries to be protected with different passwords. The `KeyStoreBuilderParameters` class then can be used to initialize a `KeyManagerFactory` using one or more of these `Builder` objects.

A `X509KeyManager` implementation in the SunJSSE provider called NewSunX509 supports these parameters. If multiple certificates are available, it attempts to pick a certificate with the appropriate key usage and prefers valid to expired certificates.

Example 8-17 illustrates how to tell JSSE to use both a PKCS#11 keystore (which might in turn use a smartcard) and a PKCS#12 file-based keystore.

Example 8-17 Sample Code to Use PKCS#11 and PKCS#12 File-based Keystore

```
import javax.net.ssl.*;
import java.security.KeyStore.*;
// ...

// Specify keystore builder parameters for PKCS#11 keystores
Builder scBuilder = Builder.newInstance("PKCS11", null,
    new CallbackHandlerProtection(myGuiCallbackHandler));

// Specify keystore builder parameters for a specific PKCS#12 keystore
Builder fsBuilder = Builder.newInstance("PKCS12", null,
    new File(pkcsFileName), new PasswordProtection(pkcsKsPassword));

// Wrap them as key manager parameters
ManagerFactoryParameters ksParams = new KeyStoreBuilderParameters(
    Arrays.asList(new Builder[] { scBuilder, fsBuilder }) );
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
// Create KeyManagerFactory
KeyManagerFactory factory = KeyManagerFactory.getInstance("NewSunX509");

// Pass builder parameters to factory
factory.init(ksParams);

// Use factory
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(factory.getKeyManagers(), null, null);
```

The PKCS12 Keystore Format

The PKCS#12 (Personal Information Exchange Syntax Standard) specifies a portable format for storage and/or transport of a user's private keys, certificates, miscellaneous secrets, and other items. The SunJSSE provider supplies a complete implementation of the PKCS12 `java.security.KeyStore` format for reading and writing PKCS12 files. This format is also supported by other toolkits and applications for importing and exporting keys and certificates, such as Mozilla Firefox, Microsoft Internet Explorer, and OpenSSL. For example, these implementations can export client certificates and keys into a file using the .p12 file name extension.

With the SunJSSE provider, you can access PKCS12 keys through the `KeyStore` API with a keystore type of PKCS12. In addition, you can list the installed keys and associated certificates by using the `keytool` command with the `-storetype` option set to `pkcs12`. See `keytool` in *Java Development Kit Tool Specifications*.

Several security properties enable you to configure how PKCS12 keystores are created. These properties begin with the prefix `keystore.pkcs12`. See the `java.security` file for more information.

Server Name Indication (SNI) Extension

The SNI extension is a feature that extends the TLS/DTLS protocol to indicate what server name the client is attempting to connect to during handshaking. Servers can use server name indication information to decide if specific `SSLSocket` or `SSLEngine` instances should accept a connection. For example, when multiple virtual or name-based servers are hosted on a single underlying network address, the server application can use SNI information to determine whether this server is the exact server that the client wants to access. Instances of this class can be used by a server to verify the acceptable server names of a particular type, such as host names. See section 3 of TLS Extensions (RFC 6066).

Developers of client applications can explicitly set the server name indication using the `SSLParameters.setServerNames(List<SNIServerName> serverNames)` method. See Example 8-18.

Developers of server applications can use the `SNIMatcher` class to decide how to recognize server name indication. Example 8-19 and Example 8-20 illustrate this functionality:

Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 8-18 Sample Code to Set the Server Name Indication

The following code sample illustrates how to set the server name indication using the method `SSLParameters.setServerNames(List<SNIServerName> serverNames)`:

```
SSLSocketFactory factory = ...
SSLSocket sslSocket = factory.createSocket("172.16.10.6", 443);
// SSLEngine sslEngine = sslContext.createSSLEngine("172.16.10.6", 443);

SNIHostName serverName = new SNIHostName("www.example.com");
List<SNIServerName> serverNames = new ArrayList<>(1);
serverNames.add(serverName);

SSLParameters params = sslSocket.getSSLParameters();
params.setServerNames(serverNames);
sslSocket.setSSLParameters(params);
// sslEngine.setSSLParameters(params);
```

Example 8-19 Sample Code Using SSLSocket Class to Recognize SNI

The following code sample illustrates how the server applications can use the `SNIMatcher` class to decide how to recognize server name indication:

```
SSLSocket sslSocket = sslServerSocket.accept();

SNIMatcher matcher = SNIHostName.createSNIMatcher("www\\.example\\.(com|
org)");
Collection<SNIMatcher> matchers = new ArrayList<>(1);
matchers.add(matcher);

SSLParameters params = sslSocket.getSSLParameters();
params.setSNIMatchers(matchers);
sslSocket.setSSLParameters(params);
```

Example 8-20 Sample Code Using SSLServerSocket Class to Recognize SNI

The following code sample illustrates how the server applications can use the `SNIMatcher` class to decide how to recognize server name indication:

```
SSLServerSocket sslServerSocket = ...;

SNIMatcher matcher = SNIHostName.createSNIMatcher("www\\.example\\.(com|
org)");
Collection<SNIMatcher> matchers = new ArrayList<>(1);
matchers.add(matcher);

SSLParameters params = sslServerSocket.getSSLParameters();
params.setSNIMatchers(matchers);
sslServerSocket.setSSLParameters(params);

SSLSocket sslSocket = sslServerSocket.accept();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The following list provides examples for the behavior of the `SNIMatcher` when receiving various server name indication requests in the ClientHello message:

- Matcher configured to `www\\.example\\.com`:
  - If the requested host name is `www.example.com`, then it will be accepted and a confirmation will be sent in the ServerHello message.
  - If the requested host name is `www.example.org`, then it will be rejected with an `unrecognized_name` fatal error.
  - If there is no requested host name or it is empty, then the request will be accepted but no confirmation will be sent in the ServerHello message.
- Matcher configured to `www\\.invalid\\.com`:
  - If the requested host name is `www.example.com`, then it will be rejected with an `unrecognized_name` fatal error.
  - If the requested host name is `www.example.org`, then it will be accepted and a confirmation will be sent in the ServerHello message.
  - If there is no requested host name or it is empty, then the request will be accepted but no confirmation will be sent in the ServerHello message.
- Matcher is not configured:

  Any requested host name will be accepted but no confirmation will be sent in the ServerHello message.

For descriptions of new classes that implement the SNI extension, see:

- StandardConstants Class
- SNIServerName Class
- SNIMatcher Class
- SNIHostName Class

For examples, see Using the Server Name Indication (SNI) Extension.

TLS Application Layer Protocol Negotiation

Negotiate an application protocol for a TLS connection with Application Layer Protocol Negotiation (ALPN).

What is ALPN?

Some applications might want or need to negotiate a shared application level value before a TLS handshake has completed. For example, HTTP/2 uses the Application Layer Protocol Negotiation mechanism to help establish which HTTP version ("h2", "spdy/3", "http/1.1") can or will be used on a particular TCP or UDP port. ALPN (RFC 7301) does this without adding network round-trips between the client and the server. In the case of HTTP/2 the protocol must be established before the connection is negotiated, as client and server need to know what version of HTTP to use before they start communicating. Without ALPN it would not be possible to have application protocols HTTP/1 and HTTP/2 on the same port.

The client uses the ALPN extension at the beginning of the TLS handshake to send a list of supported application protocols to the server as part of the `ClientHello`. The server reads the list of supported application protocols in the `ClientHello`, and determines which of the supported protocols it prefers. It then sends a `ServerHello` message back to the client with the

Copyright © 1993, 2026, Oracle and/or its affiliates.

negotiation result. The message may contain either the name of the protocol that has been chosen or that no protocol has been chosen.

The application protocol negotiation can thus be accomplished within the TLS handshake, without adding network round-trips, and allows the server to associate a different certificate with each application protocol, if desired.

Unlike many other TLS extensions, this extension does not establish properties of the session, only of the connection. That's why you'll find the negotiated values in the `SSLSocket`/`SSLEngine`, not the `SSLSession`. When session resumption or session tickets are used (see TLS Session Resumption without Server-Side State), the previously negotiated values are irrelevant, and only the values in the new handshake messages are considered.

Setting up ALPN on the Client

Set up Application Layer Protocol Negotiation (ALPN) values supported by the client to send to the server by calling the `SSLParameters.setApplicationProtocols(String[])` method, followed by the `setSSLParameters` method of either `SSLSocket` or `SSLEngine`. During the handshake with the server, the server will read the client's list of application protocols and will determine which is the most suitable.

Example 8-21 Sample Code for Setting and Getting ALPN Values in a Java Client

For example, here are the steps to set ALPN values of `"three"` and `"two"` on the client.

To run the code the property `javax.net.ssl.trustStore` must be set to a valid root certificate. (This can be done on the command line).

```
import java.io.*;
import java.util.*;
import javax.net.ssl.*;
public class SSLClient {
    public static void main(String[] args) throws Exception {

        // Code for creating a client side SSLSocket
        SSLSocketFactory sslsf = (SSLSocketFactory)
SSLSocketFactory.getDefault();
        SSLSocket sslSocket = (SSLSocket) sslsf.createSocket("localhost",
9999);

        // Get an SSLParameters object from the SSLSocket
        SSLParameters sslp = sslSocket.getSSLParameters();

        // Populate SSLParameters with the ALPN values
        // On the client side the order doesn't matter as
        // when connecting to a JDK server, the server's list takes priority
        String[] clientAPs = {"three", "two"};
        sslp.setApplicationProtocols(clientAPs);

        // Populate the SSLSocket object with the SSLParameters object
        // containing the ALPN values
        sslSocket.setSSLParameters(sslp);

        sslSocket.startHandshake();

        // After the handshake, get the application protocol that has been
negotiated
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        String ap = sslSocket.getApplicationProtocol();
        System.out.println("Application Protocol client side: \"" + ap +
"\"");

        // Do simple write/read
        InputStream sslIS = sslSocket.getInputStream();
        OutputStream sslOS = sslSocket.getOutputStream();
        sslOS.write(280);
        sslOS.flush();
        sslIS.read();
        sslSocket.close();
    }
}
```

When this code is run, it sends a `ClientHello` message to a Java server that has set the ALPN values `one`, `two`, and `three`. The code prints the following output:

```
Application Protocol client side: two
```

It is also possible to check the results of the negotiation during handshaking. See Determining Negotiated ALPN Value during Handshaking.

Setting up Default ALPN on the Server

Use the default ALPN mechanism to determine a suitable application protocol by setting ALPN values on the server.

To use the default mechanism for ALPN on the server, populate an `SSLParameters` object with the ALPN values you wish to set, and then use this `SSLParameters` object to populate either the `SSLSocket` object or the `SSLEngine` object with these parameters as you have done when you set up ALPN on the client (see the section Setting up ALPN on the Client). The first value of the ALPN values set on the server that matches any of the ALPN values contained in the `ClientHello` will be chosen and returned to the client as part of the `ServerHello`.

Example 8-22 Sample Code for Default ALPN Value Negotiation on the Server

Here is the code for a Java server that uses the default approach for protocol negotiation. To run the code the property `javax.net.ssl.keyStore` must be set to a valid keystore. (This can be done on the command line, see Creating a Keystore to Use with JSSE).

```
import java.util.*;
import javax.net.ssl.*;
public class SSLServer {
    public static void main(String[] args) throws Exception {

        // Code for creating a server side SSLSocket
        SSLServerSocketFactory sslssf =
            (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
        SSLServerSocket sslServerSocket =
            (SSLServerSocket) sslssf.createServerSocket(9999);
        SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();

        // Get an SSLParameters object from the SSLSocket
        SSLParameters sslp = sslSocket.getSSLParameters();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            // Populate SSLParameters with the ALPN values
            // As this is server side, put them in order of preference
            String[] serverAPs ={ "one", "two", "three" };
            sslp.setApplicationProtocols(serverAPs);

            // If necessary at any time, get the ALPN values set on the
            // SSLParameters object with:
            // String serverAPs = sslp.setApplicationProtocols();

            // Populate the SSLSocket object with the ALPN values
            sslSocket.setSSLParameters(sslp);

            sslSocket.startHandshake();

            // After the handshake, get the application protocol that
            // has been negotiated

            String ap = sslSocket.getApplicationProtocol();
            System.out.println("Application Protocol server side: \"" + ap +
"\"");

            // Continue with the work of the server
            InputStream sslIS = sslSocket.getInputStream();
            OutputStream sslOS = sslSocket.getOutputStream();
            sslIS.read();
            sslOS.write(85);
            sslOS.flush();
            sslSocket.close();
        }
}
```

When this code is run and a Java client sends a `ClientHello` with ALPN values `three` and `two`, the output is:

```
Application Protocol server side: two
```

It is also possible to check the results of the negotiation during handshaking. See Determining Negotiated ALPN Value during Handshaking.

Setting up Custom ALPN on the Server

Use the custom ALPN mechanism to determine a suitable application protocol by setting up a callback method.

If you do not want to use the server's default negotiation protocol, you can use the `setHandshakeApplicationProtocolSelector` method of `SSLEngine` or `SSLSocket` to register a `BiFunction` (lambda) callback that can examine the handshake state so far, and then make your selection based on the client's list of application protocols and any other relevant information. For example, you may consider using the cipher suite suggested, or the Server Name Indication (SNI) or any other data you can obtain in making the choice. If custom negotiation is used, the values set by the `setApplicationProtocols` method (default negotiation) will be ignored.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 8-23 Sample Code for Custom ALPN Value Negotiation on the Server

Here is the code for a Java server that uses the custom mechanism for protocol negotiation. To run the code the property `javax.net.ssl.keyStore` must be set to a valid certificate. (This can be done on the command line, see Creating a Keystore to Use with JSSE).

```
import java.util.*;
import javax.net.ssl.*;

public class SSLServer {
    public static void main(String[] args) throws Exception {

        // Code for creating a server side SSLSocket

        SSLServerSocketFactory sslssf =
            (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
        SSLServerSocket sslServerSocket =
            (SSLServerSocket) sslssf.createServerSocket(9999);
        SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();

        // Code to set up a callback function
        // Pass in the current SSLSocket to be inspected and client AP values

        sslSocket.setHandshakeApplicationProtocolSelector(
            (serverSocket, clientProtocols) -> {
                SSLSession handshakeSession =
serverSocket.getHandshakeSession();

                // callback function called with current SSLSocket and client
AP values
                // plus any other useful information to help determine
appropriate
                // application protocol. Here the protocol and ciphersuite
are also
                // passed to the callback function.

                return chooseApplicationProtocol(
                    serverSocket,
                    clientProtocols,
                    handshakeSession.getProtocol(),
                    handshakeSession.getCipherSuite());
         });

        sslSocket.startHandshake();

        // After the handshake, get the application protocol that has been
        // returned from the callback method.

        String ap = sslSocket.getApplicationProtocol();
        System.out.println("Application Protocol server side: \"" + ap +
"\"");

        // Continue with the work of the server

        InputStream sslIS = sslSocket.getInputStream();
        OutputStream sslOS = sslSocket.getOutputStream();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            sslIS.read();
            sslOS.write(85);
            sslOS.flush();
            sslSocket.close();
        }

        // The callback method. Note how the parameters match the call within
        // the setHandshakeApplicationProtocolSelector method.

        public static String chooseApplicationProtocol(SSLSocket serverSocket,
                List<String> clientProtocols, String protocol, String
cipherSuite ) {
            // For example, check the cipher suite and return an application
protocol
            // value based on that.
            if (cipherSuite.equals("<--a_particular_ciphersuite-->")) {
                return "three";
            } else {
                return "";
            }
        }
}
```

If the cipher suite matches the one you specify in the condition statement when this code is run , then the value `three` will be returned. Otherwise an empty string will be returned.

Note that the `BiFunction` object's return value is a `String`, which will be the application protocol name, or null to indicate that none of the advertised names are acceptable. If the return value is an empty `String` then application protocol indications will not be used. If the return value is null (no value chosen) or is a value that was not advertised by the peer, the underlying protocol will determine what action to take. (For example, the server code will send a "no_application_protocol" alert and terminate the connection.)

After handshaking completes on both client and server, you can check the result of the negotiation by calling the `getApplicationProtocol` method on either the `SSLSocket` object or the `SSLEngine` object.

Determining Negotiated ALPN Value during Handshaking

To determine the ALPN value that has been negotiated during the handshaking, create a custom `KeyManager` or `TrustManager` class, and include in this custom class a call to the `getHandshakeApplicationProtocol` method.

There are some use cases where the selected ALPN and SNI values will affect the choices made by a `KeyManager` or `TrustManager`. For example, an application might want to select different certificate/private key sets depending on the attributes of the server and the chosen ALPN/SNI/ciphersuite values.

The sample code given illustrates how to call the `getHandshakeApplicationProtocol` method from within a custom `X509ExtendedKeyManager` that you create and register as the `KeyManager` object.

Example 8-24 Sample Code for a Custom KeyManager

This example shows the entire code for a custom `KeyManager` that extends `X509ExtendedKeyManager`. Most methods simply return the value returned from the `KeyManager` class that is being wrapped by this `MyX509ExtendedKeyManager` class. However the

Copyright © 1993, 2026, Oracle and/or its affiliates.

`chooseServerAlias` method calls the `getHandshakeApplicationProtocol` on the `SSLSocket` object and therefore can determine the current negotiated ALPN value.

```
import java.net.Socket;
import java.security.*;
import javax.net.ssl.*;

public class MyX509ExtendedKeyManager extends X509ExtendedKeyManager {

    // X509ExtendedKeyManager is an abstract class so your new class
    // needs to implement all the abstract methods in this class.
    // The easiest way to do this is to wrap an existing KeyManager
    // and call its methods for each of the methods you need to implement.

    X509ExtendedKeyManager akm;

    public MyX509ExtendedKeyManager(X509ExtendedKeyManager akm) {
        this.akm = akm;
    }

    @Override
    public String[] getClientAliases(String keyType, Principal[] issuers) {
        return akm.getClientAliases(keyType, issuers);
    }

    @Override
    public String chooseClientAlias(String[] keyType, Principal[] issuers,
        Socket socket) {
        return akm.chooseClientAlias(keyType, issuers, socket);
    }

    @Override
    public String chooseServerAlias(String keyType, Principal[] issuers,
        Socket socket) {

        // This method has access to a Socket, so it is possible to call the
        // getHandshakeApplicationProtocol method here. Note the cast from
        // a Socket to an SSLSocket
        String ap = ((SSLSocket) socket).getHandshakeApplicationProtocol();
        System.out.println("In chooseServerAlias, ap is: " + ap);
        return akm.chooseServerAlias(keyType, issuers, socket);
    }

    @Override
    public String[] getServerAliases(String keyType, Principal[] issuers) {
        return akm.getServerAliases(keyType, issuers);
    }

    @Override
    public X509Certificate[] getCertificateChain(String alias) {
        return akm.getCertificateChain(alias);
    }

    @Override
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    public PrivateKey getPrivateKey(String alias) {
        return akm.getPrivateKey(alias);
    }
}
```

When this code is registered as the `KeyManager` for a Java server and a Java client sends a `ClientHello` with ALPN values, the output will be:

```
    In chooseServerAlias, ap is: <negotiated value>
```

Example 8-25 Sample Code for Using a Custom KeyManager in a Java Server

This example shows a simple Java server that uses the default ALPN negotiation strategy and the custom `KeyManager`, `MyX509ExtendedKeyManager`, shown in the prior code sample.

```
import java.io.*;
import java.util.*;
import javax.net.ssl.*;
import java.security.KeyStore;

public class SSLServerHandshake {

    public static void main(String[] args) throws Exception {
        SSLContext ctx = SSLContext.getInstance("TLS");

        // You need to explicitly create a create a custom KeyManager

        // Keystores
        KeyStore keyKS = KeyStore.getInstance("PKCS12");
        keyKS.load(new FileInputStream("serverCert.p12"),
            "password".toCharArray());

        // Generate KeyManager
        KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
        kmf.init(keyKS, "password".toCharArray());
        KeyManager[] kms = kmf.getKeyManagers();

        // Code to substitute MyX509ExtendedKeyManager
        if (!(kms[0] instanceof X509ExtendedKeyManager)) {
            throw new Exception("kms[0] not X509ExtendedKeyManager");
        }

        // Create a new KeyManager array and set the first index
        // of the array to an instance of MyX509ExtendedKeyManager.
        // Notice how creating this object is done by passing in the
        // existing default X509ExtendedKeyManager
        kms = new KeyManager[] {
            new MyX509ExtendedKeyManager((X509ExtendedKeyManager) kms[0])};

        // Initialize SSLContext using the new KeyManager
        ctx.init(kms, null, null);

        // Instead of using SSLServerSocketFactory.getDefault(),
        // get a SSLServerSocketFactory based on the SSLContext
        SSLServerSocketFactory sslssf = ctx.getServerSocketFactory();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            SSLServerSocket sslServerSocket =
                (SSLServerSocket) sslssf.createServerSocket(9999);
            SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();
            SSLParameters sslp = sslSocket.getSSLParameters();
            String[] serverAPs ={"one","two","three"};
            sslp.setApplicationProtocols(serverAPs);
            sslSocket.setSSLParameters(sslp);
            sslSocket.startHandshake();

            String ap = sslSocket.getApplicationProtocol();
            System.out.println("Application Protocol server side: \"" + ap +
"\"");

            InputStream sslIS = sslSocket.getInputStream();
            OutputStream sslOS = sslSocket.getOutputStream();
            sslIS.read();
            sslOS.write(85);
            sslOS.flush();

            sslSocket.close();
            sslServerSocket.close();
        }
}
```

With the custom `X509ExtendedKeyManager` in place, when `chooseServerAlias` is called during handshaking the `KeyManager` has the opportunity to examine the negotiated application protocol value. In the case of the example shown, this value is output to the console.

For example, when this code is run and a Java client sends a `ClientHello` with ALPN values `three` and `two`, the output will be:

```
Application Protocol server side: two
```

Reading and Writing ALPN Values with the SunJSSE Provider

ALPN transports data with byte arrays, which means that it expects text to be encoded with single byte character encodings such as US-ASCII. Java ALPN APIs use the String class for text, but prior to Java SE 16/11.0.2/8u301, the SunJSSE provider converts `String` instances to byte arrays with UTF-8. However, UTF-8 is a variable-width character encoding. It encodes characters above U+007F with more than one byte, which may not be expected by an ALPN peer.

In Java SE 16/11.0.2/8u301 and later, the SunJSSE provider encodes and decodes String characters as 8-bit ISO_8859_1/LATIN-1 characters.

ALPN values are now represented using the network byte representation expected by the peer, which should require no modification for standard 7-bit ASCII-based `String` instances.

The methods in the `javax.net.ssl.SSLSocket` and `javax.net.ssl.SSLEngine` return `ApplicationProtocol String` values in the network byte representation sent by the peer.

However, if you have Unicode data with characters that are above U+007F, then your application must correctly encode or decode them to byte arrays before sending or receiving them instead of relying on the SunJSSE provider to automatically encode or decode Unicode characters. Alternatively, you can set the security property `jdk.tls.alpnCharset` to UTF-8 to revert to the previous behavior.

Copyright © 1993, 2026, Oracle and/or its affiliates.

To compare ALPN values with their expected values, you can convert them to byte arrays and then compare them.

The expected ALPN values in the following example are the string `http/1.1` and the UTF-8 encoded string (in hexadecimal) `0xABCD0xABCE0xABCF` (which are the Meetei Mayek letters "HUK UN I"). The example converts the ALPN value to a byte array with ISO-8859-1, converts `http/1.1` to a byte array with UTF-8, and manually specifies the byte array representation of `0xABCD0xABCE0xABCF`.

```
// Get the ALPN value negotiated by the TLS handshake currently
// in progress

String networkString = sslEngine.getHandshakeApplicationProtocol();

// Encode the ALPN value into a byte array with the ISO-8859-1
// character encoding

byte[] bytes = networkString.getBytes(StandardCharsets.ISO_8859_1);

String HTTP1_1 = "http/1.1";

// Encode the String "http/1.1" into a byte array with the
// UTF-8 character set

byte[] HTTP1_1_BYTES = HTTP1_1.getBytes(StandardCharsets.UTF_8);

// Create a byte array representing the Unicode characters 0xABCD,
// 0xABCE, and 0xABCF, which are the Meetei Mayek letters "HUK UN I"

byte[] HUK_UN_I_BYTES = new byte[] {
    (byte) 0xab, (byte) 0xcd,
    (byte) 0xab, (byte) 0xce,
    (byte) 0xab, (byte) 0xcf};

// Test whether the APLN value is equal to "http/1.1" or
// 0xABCD0xABCE0xABCF

if ((Arrays.compare(bytes, HTTP1_1_BYTES) == 0 ) ||
    Arrays.compare(bytes, HUK_UN_I_BYTES) == 0) {
    // ...
}
```

Alternatively, you can compare ALPN values with the method `String.equals()` if you know that the ALPN value was encoded from a `String` using a certain character set, for example UTF-8. You must decode the ALPN value to a Unicode `String` before comparing it.

```
String unicodeString = new String(bytes, StandardCharsets.UTF_8);
if (unicodeString.equals(HTTP1_1) ||
    unicodeString.equals("\uabcd\uabce\uabcf")) {
    // ...
}
```

For the method `javax.net.ssl.SSLParameters.setApplicationProtocols(String[] protocols)`, you must convert its `String` arguments to the network byte representation

Copyright © 1993, 2026, Oracle and/or its affiliates.

expected by the peer. For example, if the peer expects ALPN values in UTF-8, you must convert it to a byte array with UTF-8 and then store it as a byte-oriented `String`:

```
// Convert Meetei Mayek letters "HUK UN I" (in hexadecimal,
0xABCD0xABCE0xABCF)
// to a byte array with UTF-8
byte[] bytes = "\uabcd\uabce\uabcf".getBytes(StandardCharsets.UTF_8);

// Create a byte-oriented String with ISO-8859-1
String HUK_UN_I = new String(bytes, StandardCharsets.ISO_8859_1);

// GREASE value {0x8A, 0x8A}
String rfc7301Grease8A = "\u008A\u008A";
SSLParameters p = sslSocket.getSSLParameters();
p.setApplicationProtocols(new String[] {"h2", "http/1.1", rfc7301Grease8A,
HUK_UN_I});
sslSocket.setSSLParameters(p);
```

At the beginning of the TLS handshake, the client sends a list of ALPN values to the server, and the server selects which values it can use and ignores those that it doesn't recognize. However, a flawed TLS implementation might instead reject unrecognized ALPN values, which may prevent the handshake from proceeding, but developers or administrators may not notice this flaw because it will still enable clients and servers whose ALPN values it recognizes to connect.

Consequently, the TLS specification has introduced Generate Random Extensions And Sustain Extensibility (GREASE) values: a reserved set of TLS protocol values that a TLS implementation may randomly advertise to ensure that peers correctly handle unrecognized values.

In the previous example, one of the values passed to the method `setApplicationProtocols`, `rfc7301Grease8A`, is a GREASE value. The peer should ignore it instead of reject it.

ALPN Related Classes and Methods

These classes and methods are used when working with Application Layer Protocol Negotiation (ALPN).

Classes and Methods to Use

`SSLEngine` and `SSLSocket` contain the same ALPN related methods and they have the same functionality.

Class	Method	Purpose
`SSLParameters`	`public String[] getApplicationProtocols();`	**Client-side and server-side**: use the method to return a `String` array containing each protocol set.
`SSLParameters`	`public void setApplicationProtocols([] protocols) ;`	**Client-side**: use the method to set the protocols that can be chosen by the server. **Server-side**: use the method to set the protocols that the server can use. The String array should contain the protocols in order of preference.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Class	Method	Purpose
`SSLEngine` `SSLSocket`	`public String getApplicationProtocol();`	**Client-side and server-side**: use the method *after* TLS protocol negotiation has completed to return a `String` containing the protocol that has been chosen for the connection.
`SSLEngine` `SSLSocket`	`public String getHandshakeApplicationProtocol();`	**Client-side and server-side**: use the method *during* handshaking to return a `String` containing the protocol that has been chosen for the connection. If this method is called before or after handshaking, it will return null. See Determining Negotiated ALPN Value during Handshaking for instructions on how to call this method.
`SSLEngine` `SSLSocket`	`public void setHandshakeApplicationProtocolSelector(BiFunction,String> selector)`	**Server-side**: use the method to register a callback function. The application protocol value can then be set in the callback based on any information available, for example the protocol or cipher suite. See Setting up Custom ALPN on the Server for instructions on how to use this method.

Troubleshooting JSSE

This section contains information for troubleshooting JSSE. First, it provides some common Configuration Problems and ways to solve them, and then it describes helpful Debugging Utilities.

Configuration Problems

Solutions to some common configuration problems.

SSLHandshakeException: No Available Authentication Scheme, Handshake Failure

Problem: The server throws this exception:

```
javax.net.ssl.SSLHandshakeException: No available authentication scheme
```

The client then receives a fatal alert:

```
javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure
```

Cause: The server throws this `SSLHandshakeException` if TLSv1.3 is chosen as the protocol version and only DSA certificates are available in the server's keymanager. Verify this with the keytool command; change `testkeys.dsa` to the name of your keystore:

```
keytool -list -keystore testkeys.dsa -v

Enter keystore password:
Keystore type: PKCS12
Keystore provider: SUN
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
Your keystore contains 1 entry

Alias name: localhost
Creation date: Sep 19, 2018
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]: Owner: CN=localhost, OU=Widget, O=Ficticious, L=Sunnyvale,
ST=CA, C=US Issuer: CN=localhost, OU=Widget, O=Ficticious, L=Sunnyvale,
ST=CA, C=US

...deleted...

Signature algorithm name: SHA256withDSA
Subject Public Key Algorithm: 2048-bit DSA key

...deleted...
```

Solution: Update your certificates so that they contain RSA or EC public keys.

CertificateException While Handshaking

Problem: When negotiating an TLS/DTLS connection, the client or server throws a `CertificateException`.

Cause 1: This is generally caused by the remote side sending a certificate that is unknown to the local side.

Solution 1: The best way to debug this type of problem is to turn on debugging (see Debugging Utilities) and watch as certificates are loaded and when certificates are received via the network connection. Most likely, the received certificate is unknown to the trust mechanism because the wrong trust file was loaded.

Refer to the following sections:

- JSSE Classes and Interfaces
- TrustManager Interface
- KeyManager Interface

Cause 2: The system clock is not set correctly. In this case, the perceived time may be outside the validity period on one of the certificates, and unless the certificate can be replaced with a valid one from a truststore, the system must assume that the certificate is invalid, and therefore throw the exception.

Solution 2: Correct the system clock time.

Runtime Exception: SSL Service Not Available

Problem: When running a program that uses JSSE, an exception occurs indicating that an SSL service is not available. For example, an exception similar to one of the following is thrown:

```
Exception in thread "main" java.net.SocketException:
    no SSL Server Sockets
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
Exception in thread "main":
    SSL implementation not available
```

Cause: There was a problem with `SSLContext` initialization, for example, due to an incorrect password on a keystore or a corrupted keystore (a JDK vendor once shipped a keystore in an unknown format, and that caused this type of error).

Solution: Check initialization parameters. Ensure that any keystores specified are valid and that the passwords specified are correct. One way that you can check this is by trying to use `keytool` to examine the keystores and the relevant contents. See `keytool` in *Java Development Kit Tool Specifications*.

Runtime Exception: "No available certificate corresponding to the SSL cipher suites which are enabled"

Problem: When trying to run a simple SSL server program, the following exception is thrown:

```
Exception in thread "main" javax.net.ssl.SSLException:
    No available certificate corresponding to the SSL cipher suites which
are enabled...
```

Cause: Various cipher suites require certain types of key material. For example, if an RSA cipher suite is enabled, then an RSA `keyEntry` must be available in the keystore. If no such key is available, then this cipher suite cannot be used. This exception is thrown if there are no available key entries for all of the cipher suites enabled.

Solution: Create key entries for the various cipher suite types, or use an anonymous suite. Anonymous cipher suites are inherently dangerous because they are vulnerable to MITM (man-in-the-middle) attacks. See RFC 5246: The Transport Layer Security (TLS) Protocol, Version 1.2.

Refer to the following sections to learn how to pass the correct keystore and certificates:

- JSSE Classes and Interfaces
- Customizing the Default Keystores and Truststores, Store Types, and Store Passwords
- The PKCS12 Keystore Format

Runtime Exception: No Cipher Suites in Common

Problem 1: When handshaking, the client and/or server throw this exception.

Cause 1: Both sides of a TLS connection must agree on a common cipher suite. If the intersection of the client's cipher suite set with the server's cipher suite set is empty, then you will see this exception.

Solution 1: Configure the enabled cipher suites to include common cipher suites, and be sure to provide an appropriate `keyEntry` for asymmetric cipher suites. Also see Runtime Exception: "No available certificate corresponding to the SSL cipher suites which are enabled" in this section.)

Problem 2: When using Mozilla Firefox or Microsoft Internet Explorer to access files on a server that only has DSA-based certificates, a runtime exception occurs indicating that there are no cipher suites in common.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Cause 2: By default, `keyEntries` created with `keytool` use DSA public keys. If only DSA `keyEntries` exist in the keystore, then only DSA-based cipher suites can be used. By default, Firefox and Internet Explorer send only RSA-based cipher suites. Because the intersection of client and server cipher suite sets is empty, this exception is thrown.

Solution 2: To interact with Firefox or Internet Explorer, you should create certificates that use RSA-based keys. To do this, specify the `-keyalg` RSA option when using keytool. For example:

```
keytool -genkeypair -alias duke -keystore testkeys -keyalg rsa
```

Socket Disconnected After Sending ClientHello Message

Problem: A socket attempts to connect, sends a ClientHello message, and is immediately disconnected.

Cause: Some TLS servers will disconnect if a ClientHello message is received in a format they do not understand or with a protocol version number that they do not support.

Solution: Try adjusting the enabled protocols on the client side. This involves modifying or invoking some of the following system properties and methods:

- System property `https.protocols` for the `HttpsURLConnection` class
- System property `jdk.tls.client.protocols`
- `SSLContext.getInstance` method
- `SSLEngine.setEnabledProtocols` method
- `SSLSocket.setEnabledProtocols` method
- `SSLParameters.setProtocols` and `SSLEngine.setSSLParameters` methods
- `SSLParameters.setProtocols` and `SSLSocket.setSSLParameters` methods

For backwards compatibility, some TLS implementations (such as SunJSSE) can send TLS ClientHello messages encapsulated in the SSLv2 ClientHello format. The SunJSSE provider supports this feature. If you want to use this feature, add the "SSLv2Hello" protocol to the enabled protocol list, if necessary. (See Protocols in the JDK Providers, which lists the protocols that are enabled by default for the SunJSSE provider.)

The TLS RFC standards require that implementations negotiate to the latest version both sides speak, but some non-conforming implementation simply hang up if presented with a version they don't understand. For example, some older server implementations that speak only SSLv3 will shutdown if TLSv1.2 is requested. In this situation, consider using a TLS version fallback scheme:

1. Fall back from TLSv1.2 to TLSv1.1 if the server does not understand TLSv1.2.
2. Fall back from TLSv1.1 to TLSv1.0 if the previous step does not work.

For example, if the enabled protocol list on the client is TLSv1, TLSv1.1, and TLSv1.2, a typical TLS version fallback scheme may look like:

1. Try to connect to server. If server rejects the TLS connection request immediately, go to step 2.
2. Try the version fallback scheme by removing the highest protocol version (for example, TLSv1.2 for the first failure) in the enabled protocol list.
3. Try to connect to the server again. If server rejects the connection, go to step 2 unless there is no version to which the server can fall back.

Copyright © 1993, 2026, Oracle and/or its affiliates.

4. If the connection fails and SSLv2Hello is not on the enabled protocol list, restore the enable protocol list and enable SSLv2Hello. (For example, the enable protocol list should be SSLv2Hello, TLSv1, TLSv1.1, and TLSv1.2.) Start again from step 1.

> **Note**
>
> A fallback to a previous version normally means security strength downgrading to a weaker protocol. It is not suggested to use a fallback scheme unless it is really necessary, and you clearly know that the server does not support a higher protocol version.

> **Note**
>
> As part of disabling SSLv3, some servers have also disabled SSLv2Hello, which means communications with SSLv2Hello-active clients (JDK 6u95) will fail. Starting with JDK 7, SSLv2Hello default to disabled on clients, enabled on servers.

SunJSSE Cannot Find a JCA Provider That Supports a Required Algorithm and Causes a NoSuchAlgorithmException

Problem: A handshake is attempted and fails when it cannot find a required algorithm. Examples might include:

```
Exception in thread ...deleted...
    ...deleted...
    Caused by java.security.NoSuchAlgorithmException: Cannot find any
        provider supporting RSA/ECB/PKCS1Padding
```

or

```
Caused by java.security.NoSuchAlgorithmException: Cannot find any
    provider supporting AES/CBC/NoPadding
```

Cause: SunJSSE uses JCE for all its cryptographic algorithms. If the SunJCE provider has been deregistered from the `Provider` mechanism and an alternative implementation from JCE is not available, then this exception will be thrown.

Solution: Ensure that the SunJCE is available by checking that the provider is registered with the `Provider` interface. Try to run the following code in the context of your SSL connection:

```
import javax.crypto.*;

System.out.println("=====Where did you get AES=====");
Cipher c = Cipher.getInstance("AES/CBC/NoPadding");
System.out.println(c.getProvider());
```

Exception Thrown When Obtaining Application Resources from a Virtual Host Web Server that Requires an SNI Extension

Copyright © 1993, 2026, Oracle and/or its affiliates.

Problem: If you receive an `Exception` when trying to obtain application resources from your web server over TLS, and your web server is implemented as a virtual host that requires a valid Server Name Indication (SNI) extension (such as Apache HTTP Server) to distinguish the virtual host, then the web server might not be configured correctly.

Cause: Because Java SE supports the SNI extension in the JSSE client, the requested host name of the virtual server is included in the first message sent from the client to the server during the TLS handshake. The server may deny the client's request for a connection if the requested host name (the server name indication) does not match the expected server name, which should be specified in the virtual host's configuration. This triggers an TLS handshake unrecognized name alert, which results in an `Exception` being thrown.

Solution: If the cause of the problem is `javax.net.ssl.SSLProtocolException: handshake alert: unrecognized_name`, it is likely that the virtual host configuration for SNI is incorrect. If you are using Apache HTTP Server, see Name-based Virtual Host Support about configuring virtual hosts. In particular, ensure that the `ServerName` directive is configured properly in a `<VirtualHost>` block.

See the following:

- SSL with Virtual Hosts Using SNI from Apache HTTP Server Wiki
- SSL/TLS Strong Encryption: FAQ from Apache HTTP Server Documentation
- RFC 3546, Transport Layer Security (TLS) Extensions
- Bug 7194590: SSL handshaking error caused by virtual server misconfiguration

IllegalArgumentException When RC4 Cipher Suites are Configured for DTLS

Problem: An `IllegalArgumentException` exception is thrown when RC4 cipher suite algorithm is specified in `SSLEngine.setEnabledCipherSuites(String[] suites)` method and the `SSLEngine` is a DTLS engine.

```
sslContext = SSLContext.getInstance("DTLS");

// Create the engine
SSLEngine engine = sslContext.createSSLengine(hostname, port);

String enabledSuites[] = { "SSL_RSA_WITH_RC4_128_SHA" };
engine.setEnabledCipherSuites(enabledSuites);
```

Cause: According to DTLS Version 1.0 and DTLS Version 1.2, RC4 cipher suites must not be used with DTLS.

Solution: Do not use RC4 based cipher suites for DTLS connections. See "JSSE Cipher Suite Names" in Java Security Standard Algorithm Names.

Debugging Utilities

The SunJSSE provider supports dynamic debug tracing. This is similar to the mechanism that debugs security library issues. The generic Java dynamic debug tracing support is accessed with the `java.security.debug` system property, whereas the JSSE-specific dynamic debug tracing support is accessed with the `javax.net.debug` system property.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> Currently, the SunJSSE provider uses the debug utility. There is no guarantee that other providers use the debug utility. If other providers support the debug utility, then the implementation and output may be different. There is no guarantee the debug utility will continue to exist or be the same (for example, have the same options or output format) in future releases.

To view the options of the JSSE dynamic debug utility, use the following command-line option on the `java` command, where `MyApp` is an existing Java application:

```
java -Djavax.net.debug=help MyApp
```

> **Note**
>
> - The `MyApp` application will not run after the debug help information is printed, as the help code causes the application to exit.
> - If you specify the value `help` with either dynamic debug utility when running a program that does not use any classes that the utility was designed to debug, you will not get the debugging options.

The current options for the `javax.net.debug` system property are:

- `all`: Turn on all debugging
- `ssl`: Turn on SSL debugging; all SSL debug information is printed except those generated by the options `data`, `packet`, and `plaintext`
- No value: If you specify no value for the `javax.net.debug` system property, then the JSSE debug libraries log to an instance of `System.Logger` named `javax.net.ssl`.

The following can be used with the `ssl` option to select what type of debug information to print:

- `defaultctx`: Print default SSL initialization
- `handshake`: Print each handshake message
- `keygen`: Print key generation data
- `keymanager`: Print key manager tracing
- `pluggability`: Print pluggability tracing
- `record`: Enable per-record tracing
- `respmgr`: Print status response manager tracing
- `session`: Print session activity
- `sessioncache`: Print session cache tracing
- `sslctx`: Print `SSLContext` tracing
- `trustmanager`: Print trust manager tracing

Messages generated from the `handshake` option can be widened with these options:

- `data`: Hex dump of each handshake message

Copyright © 1993, 2026, Oracle and/or its affiliates.

- `verbose`: Verbose handshake message printing

Messages generated from the `record` option can be widened with these options:

- `plaintext`: Hex dump of record plaintext
- `packet`: Print raw SSL/TLS packets

To enable JSSE-specific dynamic debug tracing, set the value of the `javax.net.debug` system property (see How to Specify a java.security.Security Property) to either `all` or `ssl`. For the `ssl` option, to specify additional options, specify them after the `ssl` option. You do *not* have to have a separator between options, although a separator such as a colon (`:`) or a comma (`,`) helps readability. It does not matter what separators you use, and the ordering of the option keywords is also not important.

For an introduction to reading this debug information, see Debugging TLS Connections.

The following are examples of using the `javax.net.debug` system property:

- To view all debugging messages:

  ```
  java -Djavax.net.debug=all MyApp
  ```

- To view the hexadecimal dumps of each handshake message (the colons are optional):

  ```
  java -Djavax.net.debug=ssl:handshake:data MyApp
  ```

- To view the hexadecimal dumps of each handshake message, and to print trust manager tracing (the commas are optional):

  ```
  java -Djavax.net.debug=ssl,handshake,data,trustmanager MyApp
  ```

Debugging TLS Connections

Understanding TLS connection problems can sometimes be difficult, especially when it is not clear what messages are actually being sent and received. JSSE has a built-in debug facility and is activated by the system property `javax.net.debug`. To know more about `javax.net.debug` System property, see Debugging Utilities.

This section gives a brief overview of the debug output of the basic TLS 1.3 handshake. To know more about the TLS protocol, see RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3.

> **Note**
>
> - Debug output information about all possible TLS handshake combinations and protocols is beyond the scope of this guide. Instead, refer to the relevant RFC for more detailed information about a particular version of TLS. See TLS and DTLS Protocols for a list of supported SSL/TLS/DTLS protocols and links to their respective RFCs.
> - The output is non-standard and may change from release to release.

This example uses the default JSSE X509KeyManager and X509TrustManager, which also prints debug information about the keys and trusted certificates used during a connection. It

Copyright © 1993, 2026, Oracle and/or its affiliates.

uses the `ClassFileServer` and `SSLSocketClientWithClientAuth` sample applications from JSSE Sample Code in the Java SE 8 documentation. See `ClassFileServer` is a simple HTTPS server that can require client authentication. `SSLSocketClientWithClientAuth` demonstrates how to use the `SSLSocket` class as a client to send an HTTP request and get a response from an HTTPS server. To make things simpler, both `ClassFileServer` and `SSLSocketClientWithClientAuth` are run from the same host. See Running the JSSE Sample Code and README.txt for information about running these samples.

Run ClassFileServer on localhost

Run the `ClassFileServer` application on `localhost` using an available port. Ensure that a file named `index.html` exists in the directory specified as the root of the file server.

Run SSLSocketClientWithClientAuth on locahost

Run the `SSLSocketClientWithClientAuth` application on the same host (`localhost`) and port as `ClassFileServer`. The application connects to the HTTPS server that you started previously. It sends an HTTPS request to the server and receives the response. Ensure that you specify `-Djavax.net.debug=all` on the command line, which turns on all debugging.

Debug Output Format

Each line of the debug output contains the following information; each field is separated by a vertical bar (`|`):

- Logger name (`System.getLogger("javax.net.ssl")`)
- Debug level (`System.Logger.Level`)
- Thread ID (`Thread.currentThread().getId()`)
- Thread name (`Thread.currentThread().getName()`)
- Date and time
- Caller (location of the logging call)
- Message

Determine Client-Side and Server-Side Enabled Cipher Suites

The values of the system properties `jdk.tls.client.cipherSuites` and `jdk.tls.server.cipherSuites` are checked to determine the default enabled cipher suites; see Specifying Default Enabled Cipher Suites for more information about these system properties.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.182 EDT|
SSLContextImpl.java:443|System property jdk.tls.client.cipherSuites is set to
'null'
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.252 EDT|
SSLContextImpl.java:443|System property jdk.tls.server.cipherSuites is set to
'null'
...
```

The values of these system properties are null, so the default enabled cipher suites are those that the SunJSSE provider enables by default; see The SunJSSE Provider in JDK Providers Documentation.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The value of `jdk.tls.keyLimits` is checked to determine the limit of the amount of data an algorithm may encrypt with a specific set of keys; see Limiting Amount of Data Algorithms May Encrypt with a Set of Keys.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.334 EDT|SSLCipher.java:425|
jdk.tls.keyLimits:  entry = AES/GCM/NoPadding KeyUpdate 2^37. AES/GCM/
NOPADDING:KEYUPDATE = 137438953472
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.334 EDT|SSLCipher.java:425|
jdk.tls.keyLimits:  entry =  ChaCha20-Poly1305 KeyUpdate 2^37. CHACHA20-
POLY1305:KEYUPDATE = 137438953472
...
```

The debug output lists unsupported and disabled cipher suites:

```
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.360 EDT|
SSLContextImpl.java:431|Ignore disabled cipher suites for protocols:
[TLSv1.3, TLSv1.2]
[TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384,T
LS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256,TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384,TL
S_ECDH_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256,TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256,TL
S_ECDH_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA,TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA,TLS_ECDH
_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_AES_128_GCM_SHA256,TLS_RSA_WITH_AES_256_CBC_SHA256,TLS_RSA_WITH_A
ES_128_CBC_SHA256,TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA]
...
```

The debug output lists available cipher suites for the supported protocols:

```
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.361 EDT|
SSLContextImpl.java:431|Available cipher suites for protocols: [TLSv1.3,
TLSv1.2]
[TLS_AES_256_GCM_SHA384,TLS_AES_128_GCM_SHA256,TLS_CHACHA20_POLY1305_SHA256,TL
S_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305
_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA
256,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256,TLS_DHE_DSS_WITH_AES_256_GCM_SHA384,
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384,TL
S_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256,TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256,
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256,TLS_DHE_RSA_WITH_AES_128_CBC_SHA256,TLS_DH
E_DSS_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA,TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,TLS_EC
DHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_RSA_WITH_AES_256_CBC_SHA,TLS_DHE_DS
S_WITH_AES_256_CBC_SHA,TLS_DHE_RSA_WITH_AES_128_CBC_SHA
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
TLS_DHE_DSS_WITH_AES_128_CBC_SHA,TLS_EMPTY_RENEGOTIATION_INFO_SCSV]
...
```

Initialize X509KeyManager

The X509KeyManager is initialized. It discovers that there is one keyEntry in the supplied KeyStore for a subject called "duke". If this application wants to authenticate itself, then the X509KeyManager searches its list of keyEntries for an appropriate credential.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.618 EDT|
SunX509KeyManagerImpl.java:133|found key for : duke (
  "certificate" : {
    "version"            : "v3",
    "serial number"      : "06:ca:dd:d7:ea:fc:49:64",
    "signature algorithm": "SHA256withRSA",
    "issuer"             : "CN=Duke, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "not before"         : "2026-06-30 11:33:35.000 EDT",
    "not  after"         : "2036-06-27 11:33:35.000 EDT",
    "subject"            : "CN=Duke, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "subject public key" : "RSA",
    "extensions"         : [
      {
        ObjectId: 2.5.29.37 Criticality=false
        ExtendedKeyUsages [
          clientAuth
        ]
      },
      {
        ObjectId: 2.5.29.14 Criticality=false
        SubjectKeyIdentifier [
        KeyIdentifier [
        0000: 96 F2 A5 C0 D0 D4 EB 29   3E 85 9C 1F F4 79 CB
E2  .......)>....y..
        0010: 8A 45 05 D1                                        .E..
        ]
        ]
      }
    ]}
)
...
```

Initialize a TrustManager

A TrustManager is initialized and it finds in the truststore a self-signed certificate with a distinguished name “localhost”. A server that presents valid credentials (certificates) that chain back to a trusted certificate in the truststore will itself be trusted.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.624 EDT|
TrustStoreManager.java:113|trustStore is: ../../sampletruststore
trustStore type is: PKCS12
trustStore provider is: unspecified
the last modified time is: Tue Jun 30 11:35:43 EDT 2026
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.624 EDT|
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
TrustStoreManager.java:328|Reload the trust store
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.639 EDT|
TrustStoreManager.java:337|Reload trust certs
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.640 EDT|
TrustStoreManager.java:343|Reloaded 2 trust certs
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.661 EDT|
X509TrustManagerImpl.java:78|adding as trusted certificates (
  "certificate" : {
    "version"            : "v3",
    "serial number"      : "06:ca:dd:d7:ea:fc:49:64",
    "signature algorithm": "SHA256withRSA",
    "issuer"             : "CN=Duke, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "not before"         : "2026-06-30 11:33:35.000 EDT",
    "not  after"         : "2036-06-27 11:33:35.000 EDT",
    "subject"            : "CN=Duke, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "subject public key" : "RSA",
    "extensions"         : [
      {
        ObjectId: 2.5.29.37 Criticality=false
        ExtendedKeyUsages [
          clientAuth
        ]
      },
      {
        ObjectId: 2.5.29.14 Criticality=false
        SubjectKeyIdentifier [
        KeyIdentifier [
        0000: 96 F2 A5 C0 D0 D4 EB 29   3E 85 9C 1F F4 79 CB
E2  .......)>....y..
        0010: 8A 45 05 D1                                        .E..
        ]
        ]
      }
    ]},
  "certificate" : {
    "version"            : "v3",
    "serial number"      : "00:a6:48:9d:49:bd:64:3f:3d",
    "signature algorithm": "SHA256withRSA",
    "issuer"             : "CN=localhost, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "not before"         : "2026-06-30 11:33:18.000 EDT",
    "not  after"         : "2036-06-27 11:33:18.000 EDT",
    "subject"            : "CN=localhost, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "subject public key" : "RSA",
    "extensions"         : [
      {
        ObjectId: 2.5.29.37 Criticality=false
        ExtendedKeyUsages [
          serverAuth
        ]
      },
      {
        ObjectId: 2.5.29.17 Criticality=false
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        SubjectAlternativeName [
          DNSName: localhost
          IPAddress: 127.0.0.1
        ]
      },
      {
        ObjectId: 2.5.29.14 Criticality=false
        SubjectKeyIdentifier [
        KeyIdentifier [
        0000: 8B 39 75 65 48 73 1E C2   CE E0 D7 AE 24 93 FC
E7  .9ueHs......$...
        0010: 3F 84 65 C7                                        ?.e.
        ]
        ]
      }
    ]}
)
...
```

Perform Additional Initialization

The example performs additional initialization code, then connects to the server.

```
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.661 EDT|
SSLContextImpl.java:109|trigger seeding of SecureRandom
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.686 EDT|
SSLContextImpl.java:113|done seeding of SecureRandom
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.754 EDT|
SSLConfiguration.java:574|System property jdk.tls.client.SignatureSchemes is
set to 'null'
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.887 EDT|
HandshakeContext.java:289|Ignore unsupported cipher suite:
TLS_AES_256_GCM_SHA384 for TLSv1.2
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.887 EDT|
HandshakeContext.java:289|Ignore unsupported cipher suite:
TLS_AES_128_GCM_SHA256 for TLSv1.2
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.887 EDT|
HandshakeContext.java:289|Ignore unsupported cipher suite:
TLS_CHACHA20_POLY1305_SHA256 for TLSv1.2
...
```

The debug output also notifies you of disabled, unsupported, or unavailable extensions and signature algorithms:

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.957 EDT|SSLExtension.java:849|
System property jdk.tls.client.disableExtensions is set to 'null'
javax.net.ssl|WARNING|30|main|2026-06-30 14:21:17.959 EDT|
ServerNameExtension.java:266|Unable to indicate server name
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.959 EDT|
SSLExtensions.java:282|Ignore, context unavailable extension: server_name
javax.net.ssl|INFO|30|main|2026-06-30 14:21:17.963 EDT|AlpnExtension.java:176|
No available application protocols
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.964 EDT|
SSLExtensions.java:282|Ignore, context unavailable extension:
application_layer_protocol_negotiation
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.965 EDT|
SessionTicketExtension.java:416|Stateless resumption supported
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.965 EDT|
SignatureScheme.java:439|Ignore unsupported signature scheme: ecdsa_sha224
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.965 EDT|
SignatureScheme.java:439|Ignore unsupported signature scheme: rsa_sha224
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.966 EDT|
SignatureScheme.java:439|Ignore unsupported signature scheme: dsa_sha224
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.966 EDT|
SignatureScheme.java:458|Ignore disabled signature scheme: ecdsa_sha1
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.966 EDT|
SignatureScheme.java:458|Ignore disabled signature scheme: rsa_pkcs1_sha1
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.966 EDT|
SignatureScheme.java:458|Ignore disabled signature scheme: dsa_sha1
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.966 EDT|
SignatureScheme.java:458|Ignore disabled signature scheme: rsa_md5
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.967 EDT|
SignatureScheme.java:439|Ignore unsupported signature scheme: ecdsa_sha224
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.967 EDT|
SignatureScheme.java:439|Ignore unsupported signature scheme: rsa_sha224
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.968 EDT|
SignatureScheme.java:439|Ignore unsupported signature scheme: dsa_sha224
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:17.968 EDT|
SignatureScheme.java:458|Ignore disabled signature scheme: rsa_md5
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:17.969 EDT|
SSLExtensions.java:282|Ignore, context unavailable extension: cookie
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.008 EDT|
SSLExtensions.java:282|Ignore, context unavailable extension:
quic_transport_parameters
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.008 EDT|
SSLExtensions.java:282|Ignore, context unavailable extension:
renegotiation_info
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.008 EDT|
PreSharedKeyExtension.java:661|No session to resume.
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.009 EDT|
SSLExtensions.java:282|Ignore, context unavailable extension: pre_shared_key
...
```

Send ClientHello Message

The client sends a ClientHello message to the server. This message specifies the following:

- Client version: For TLS 1.3, this has a fixed value, TLSv1.2; TLS 1.3 uses the extension supported_versions and not this field to negotiate protocol version
- Random: A random value used to initialize the cryptographic algorithms
- Session ID: Previous versions of TLS use this ID to support a session resumption feature
- Cipher Suites: The list of cipher suites that the client requests; depending on the enabled cipher suites, there may be a broad mix of cipher suite names, some of which are only for TLSv1.3 while others are for TLSv1.2 and earlier
- Compression methods: For TLS 1.3, this must have the value 0
- Extensions:
  - status_request: The client requests OCSP; see Client-Driven OCSP and OCSP Stapling

Copyright © 1993, 2026, Oracle and/or its affiliates.

- supported_groups: Lists the named groups that the client supports for key exchange. These named groups include elliptic curve groups (ECDHE) and finite field groups (DHE). The ClientHello message must include this message if it's using ECDHE or DHE key exchange.
- ec_point_formats: Lists the elliptical curve point formats that the client can parse; in this example, the client can parse uncompressed point formats only. Other formats include compressed and ansiX962_compressed_prime.
- signature algorithms: Lists which signature algorithms may be used in CertificateVerify messages
- signature_algorithms_cert: Lists which signature algorithms may be used in digital signatures
- status_request_v2: Enables clients to specify and support several certificate status methods. Note that this extension is deprecated for TLS 1.3.
- extended_master_secret: In TLS 1.2 and earlier, this extension requests that both sides digest larger parts of the handshake transcript into the master secret than the original version of the protocol did; see RFC 7627. The extension is included in TLS 1.3 handshakes in case a TLS 1.2 handshake is negotiated.
- supported_versions: Lists which versions of TLS the client supports. In particular, if the client requests TLS 1.3, then the client version field has the value TLSv1.2 and this extension contains the value TLSv1.3; if the client requests TLS 1.2, then the client version field has the value TLSv1.2 and this extension either doesn't exist or contains the value TLSv1.2 but not the value TLSv1.3.
- psk_key_exchange_modes: Lists which key exchange modes that may be used with pre-shared keys (PSKs); in this example, the client supports PSK with (EC)DHE key establishment (psk_dhe_ke). In this mode, the client and server must supply values for the key_share extension.
- key_share: Lists cryptographic parameters for key exchange. It contains a field named client_shares that contains this list. Each item of this list contains two fields: group and key_exchange. This example contains key exchange information for the elliptical curve secp256r1.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.015 EDT|ClientHello.java:640|
Produced ClientHello handshake message (
"ClientHello": {
  "client version"      : "TLSv1.2",
  "random"              :
"CB74714B57F8A3C6CAEE09581447AF67C8E087B71D79B7D38AB1E6C7EE48301C",
  "session id"          :
"424D457B77DF36A4CEEE607539E1E0F4DE583592198FDA46F0978FF69116E286",
  "cipher suites"       : "[TLS_AES_256_GCM_SHA384(0x1302),
TLS_AES_128_GCM_SHA256(0x1301), TLS_CHACHA20_POLY1305_SHA256(0x1303),
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384(0xC02C),
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256(0xC02B),
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256(0xCCA9),
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384(0xC030),
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256(0xCCA8),
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256(0xC02F),
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384(0x009F),
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256(0xCCAA),
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384(0x00A3),
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256(0x009E),
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256(0x00A2),
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384(0xC024),
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384(0xC028),
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256(0xC023),
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256(0xC027),
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256(0x006B),
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256(0x006A),
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256(0x0067),
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256(0x0040),
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA(0xC00A),
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA(0xC014),
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA(0xC009),
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA(0xC013),
TLS_DHE_RSA_WITH_AES_256_CBC_SHA(0x0039),
TLS_DHE_DSS_WITH_AES_256_CBC_SHA(0x0038),
TLS_DHE_RSA_WITH_AES_128_CBC_SHA(0x0033),
TLS_DHE_DSS_WITH_AES_128_CBC_SHA(0x0032),
TLS_EMPTY_RENEGOTIATION_INFO_SCSV(0x00FF)]",
  "compression methods" : "00",
  "extensions"          : [
    "status_request (5)": {
      "certificate status type": ocsp
      "OCSP status request": {
        "responder_id": <empty>
        "request extensions": {
          <empty>
        }
      }
    },
    "supported_groups (10)": {
      "named groups": [X25519MLKEM768, x25519, secp256r1, secp384r1,
secp521r1, x448, ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, ffdhe8192]
    },
    "ec_point_formats (11)": {
      "formats": [uncompressed]
    },
    "status_request_v2 (17)": {
      "cert status request": {
        "certificate status type": ocsp_multi
        "OCSP status request": {
          "responder_id": <empty>
          "request extensions": {
            <empty>
          }
        }
      }
    },
    "extended_master_secret (23)": {
      <empty>
    },
    "session_ticket (35)": {
      <empty>
    },
    "signature_algorithms (13)": {
      "signature schemes": [ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384,
ecdsa_secp521r1_sha512, ed25519, ed448, rsa_pss_rsae_sha256,
rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256,
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
rsa_pss_pss_sha384, rsa_pss_pss_sha512, rsa_pkcs1_sha256, rsa_pkcs1_sha384,
rsa_pkcs1_sha512, dsa_sha256]
    },
    "supported_versions (43)": {
      "versions": [TLSv1.3, TLSv1.2]
    },
    "psk_key_exchange_modes (45)": {
      "ke_modes": [psk_dhe_ke]
    },
    "signature_algorithms_cert (50)": {
      "signature schemes": [ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384,
ecdsa_secp521r1_sha512, ed25519, ed448, rsa_pss_rsae_sha256,
rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256,
rsa_pss_pss_sha384, rsa_pss_pss_sha512, rsa_pkcs1_sha256, rsa_pkcs1_sha384,
rsa_pkcs1_sha512, dsa_sha256, ecdsa_sha1, rsa_pkcs1_sha1, dsa_sha1]
    },
    "key_share (51)": {
      "client_shares": [
        {
          "named group": X25519MLKEM768
          "key_exchange": {
            0000: B5 DC 36 AE 13 5F 95 94   54 53 E2 34 8C 5C 56
96  ..6.._..TS.4.\V.
            0010: 47 91 1E BC 6C 19 62 9F   B6 D5 24 5E E9 52 F9 47
G...l.b...$^.R.G
            0020: 39 8D 4C 39 1D B1 60 38   F2 1D 73 87 74 DE 57 5F
9.L9..`8..s.t.W_
...
          }
        },
        {
          "named group": x25519
          "key_exchange": {
            0000: F4 23 FD 3C 19 87 A7 0F   04 66 E6 43 A5 9E 7D
92  .#.<.....f.C....
            0010: B7 4D 02 8D CC 33 59 86   F2 F2 0C 23 53 6A 00
27  .M...3Y....#Sj.'
          }
        },
      ]
    }
  ]
}
)
...
```

Show Actual Data Sent and Read

The debug output shows the actual data sent to the raw output object (in this case, an OutputStream):

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.017 EDT|
SSLSocketOutputRecord.java:266|WRITE: TLSv1.3 handshake, length = 1558
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.020 EDT|
SSLSocketOutputRecord.java:281|Raw write (
  0000: 16 03 03 06 16 01 00 06   12 03 03 CB 74 71 4B 57  ............tqKW
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
  0010: F8 A3 C6 CA EE 09 58 14   47 AF 67 C8 E0 87 B7 1D  ......X.G.g.....
  0020: 79 B7 D3 8A B1 E6 C7 EE   48 30 1C 20 42 4D 45 7B  y.......H0. BME.
...
```

Then, the debug output shows the raw data read from the input device (InputStream) before any processing has been performed:

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.040 EDT|
SSLSocketInputRecord.java:501|Raw read (
  0000: 16 03 03 04 BA                                     .....
)
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.041 EDT|
SSLSocketInputRecord.java:214|READ: TLSv1.2 handshake, length = 1210
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.043 EDT|
SSLSocketInputRecord.java:501|Raw read (
  0000: 02 00 04 B6 03 03 CA E9   1B 44 80 A6 78 72 27 42  .........D..xr'B
  0010: 5D A0 EA 2C 9B BD 0E B5   0E 78 5A 4E C4 ED AB 3A  ]..,.....xZN...:
  0020: 63 11 94 C0 D8 D0 20 42   4D 45 7B 77 DF 36 A4 CE  c..... BME.w.6..
...
```

Whenever the client sends or reads a message, the debug output shows the raw data sent or read and how any messages (and their extensions) have been processed. The following sections omit these parts of the debug output.

Read ServerHello Message

At this point, TLS 1.3 has been negotiated. The server selects the TLS version and replies using a combination of the server version and the supported_versions extension. In this case, a TLSv1.3 protocol was indicated.

The ServerHello message specifies the following:

- Server version: For TLS 1.3, this must have the value TLSv1.2; TLS 1.3 uses the extension supported_versions and not this field to indicate the negotiated protocol version
- Random: Also used to initialize the cryptographic algorithms
- Session ID: For TLS 1.3, this has the same value as the corresponding field of the ClientHello message
- Cipher suite: The selected cipher suite; in this example, it is TLS_AES_128_GCM_SHA256
- Compression methods: For TLS 1.3, this must have the value 0
- Extensions
  - supported_versions: Specifies which TLS version the server uses. Note that for TLS 1.3, the server must use the value of the ClientHello message's supported_versions extension for version negotiation instead of the value of the client version field.
  - key_share: The named group and key values for a ECDHE key exchange

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.044 EDT|
SSLSocketInputRecord.java:247|READ: TLSv1.2 handshake, length = 1210
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.056 EDT|ServerHello.java:939|
Consuming ServerHello handshake message (
"ServerHello": {
  "server version"      : "TLSv1.2",
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
  "random"             :
"CAE91B4480A6787227425DA0EA2C9BBD0EB50E785A4EC4EDAB3A631194C0D8D0",
  "session id"         :
"424D457B77DF36A4CEEE607539E1E0F4DE583592198FDA46F0978FF69116E286",
  "cipher suite"       : "TLS_AES_256_GCM_SHA384(0x1302)",
  "compression methods" : "00",
  "extensions"         : [
    "supported_versions (43)": {
      "selected version": [TLSv1.3]
    },
    "key_share (51)": {
      "server_share": {
        "named group": X25519MLKEM768
        "key_exchange": {
          0000: B3 D3 E2 38 0A 95 6D 13   23 14 44 8C 69 BA 00
6A  ...8..m.#.D.i..j
          0010: DD 1F 8B 47 31 96 6D F8   8C 34 76 EA 36 C3 71
C2  ...G1.m..4v.6.q.
          0020: 2D 51 AD 11 CA DE 0E 8C   34 A2 2A 48 39 1D 38 27  -
Q......4.*H9.8'
...
```

The session is initialized:

```
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:18.073 EDT|
SSLSessionImpl.java:227|Session initialized:  Session(1782843678073|
TLS_AES_256_GCM_SHA384)
...
```

Read EncryptedExtensions Message

At this point in the handshake, enough cryptographic information has been exchanged, and the remainder of the handshake will be performed encrypted.

The EncryptedExtensions message contains responses to ClientHello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates; in this example, it returns the list of named groups that the client supports for key exchange.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.126 EDT|
EncryptedExtensions.java:172|Consuming EncryptedExtensions handshake message (
"EncryptedExtensions": [
  "supported_groups (10)": {
    "named groups": [X25519MLKEM768, x25519, secp256r1, secp384r1, secp521r1,
x448, ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, ffdhe8192]
  }
]
)
...
```

Read Server's CertificateRequest Message

The server sends the CertificateRequest message if certificate-based client authentication is desired. This message contains the desired parameters for that certificate. It specifies the following:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- certificate_request_context: A string that identifies the certificate request; the value of this field is of zero length unless it's being used for post-handshake authentication
- Extensions: The following two extensions indicate which signature algorithms may be used in digital signatures:
  - signature_algorithms: Originally appearing in TLS 1.2, applies to signatures in CertificateVerify messages
  - signature_algorithms_cert: Applies to signatures in certificates

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.130 EDT|
CertificateRequest.java:1000|Consuming CertificateRequest handshake message (
"CertificateRequest": {
  "certificate_request_context": "",
  "extensions": [
    "signature_algorithms (13)": {
      "signature schemes": [ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384,
ecdsa_secp521r1_sha512, ed25519, ed448, rsa_pss_rsae_sha256,
rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256,
rsa_pss_pss_sha384, rsa_pss_pss_sha512]
    },
    "certificate_authorities (47)": {
      "certificate authorities": [
        CN=Duke, OU=JSSE Samples, O=Demo, L=Toronto, ST=ON, C=CA
        CN=localhost, OU=JSSE Samples, O=Demo, L=Toronto, ST=ON, C=CA]
    },
    "signature_algorithms_cert (50)": {
      "signature schemes": [ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384,
ecdsa_secp521r1_sha512, ed25519, ed448, rsa_pss_rsae_sha256,
rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256,
rsa_pss_pss_sha384, rsa_pss_pss_sha512, rsa_pkcs1_sha256, rsa_pkcs1_sha384,
rsa_pkcs1_sha512, ecdsa_sha1, rsa_pkcs1_sha1]
    }
  ]
}
)
...
```

Read Server's Certificate Message

The Certificate message contains the authentication certificate and any other supporting certificates in the certificate chain. It specifies the following:

- certificate_request_context: For server authentication, this field is empty
- certificate_list: Contains a certificate chain signed by a signature algorithm advertised by the client. However, in this example, a self-signed certificate (a certificate whose subject and issue name are identical) was received. This same self-signed certificate was discovered earlier during initialization, so it will be trusted when the TrustManager is actually called to verify the received certificate.

There are many different ways of establishing trust, so if the default X509TrustManager is not doing the types of trust management you need, you can supply your own X509TrustManager to SSLContext.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.140 EDT|
CertificateMessage.java:1133|Consuming server Certificate handshake message (
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
"Certificate": {
  "certificate_request_context": "",
  "certificate_list": [
  {
    "certificate" : {
      "version"            : "v3",
      "serial number"      : "00:a6:48:9d:49:bd:64:3f:3d",
      "signature algorithm": "SHA256withRSA",
      "issuer"             : "CN=localhost, OU=JSSE Samples, O=Demo,
L=Toronto, ST=ON, C=CA",
      "not before"         : "2026-06-30 11:33:18.000 EDT",
      "not  after"         : "2036-06-27 11:33:18.000 EDT",
      "subject"            : "CN=localhost, OU=JSSE Samples, O=Demo,
L=Toronto, ST=ON, C=CA",
      "subject public key" : "RSA",
      "extensions"         : [
        {
          ObjectId: 2.5.29.37 Criticality=false
          ExtendedKeyUsages [
            serverAuth
          ]
        },
        {
          ObjectId: 2.5.29.17 Criticality=false
          SubjectAlternativeName [
            DNSName: localhost
            IPAddress: 127.0.0.1
          ]
        },
        {
          ObjectId: 2.5.29.14 Criticality=false
          SubjectKeyIdentifier [
          KeyIdentifier [
          0000: 8B 39 75 65 48 73 1E C2   CE E0 D7 AE 24 93 FC
E7  .9ueHs......$...
          0010: 3F 84 65 C7                                        ?.e.
          ]
          ]
        }
      ]}
    "extensions": {
      <no extension>
    }
  },
]
}
)
...
```

The client recognizes this certificate and can trust it.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.155 EDT|
X509TrustManagerImpl.java:298|Found trusted certificate (
  "certificate" : {
    "version"            : "v3",
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    "serial number"      : "00:a6:48:9d:49:bd:64:3f:3d",
    "signature algorithm": "SHA256withRSA",
    "issuer"             : "CN=localhost, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "not before"         : "2026-06-30 11:33:18.000 EDT",
    "not  after"         : "2036-06-27 11:33:18.000 EDT",
    "subject"            : "CN=localhost, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
    "subject public key" : "RSA",
    "extensions"         : [
      {
        ObjectId: 2.5.29.37 Criticality=false
        ExtendedKeyUsages [
          serverAuth
        ]
      },
      {
        ObjectId: 2.5.29.17 Criticality=false
        SubjectAlternativeName [
          DNSName: localhost
          IPAddress: 127.0.0.1
        ]
      },
      {
        ObjectId: 2.5.29.14 Criticality=false
        SubjectKeyIdentifier [
        KeyIdentifier [
        0000: 8B 39 75 65 48 73 1E C2   CE E0 D7 AE 24 93 FC
E7  .9ueHs......$...
        0010: 3F 84 65 C7                                        ?.e.
        ]
        ]
      }
    ]}
)
...
```

Read Server's CertificateVerify Message

The certificate sent by the server is verified by the CertificateVerify message. The message is used to provide explicit proof that the server has the private key corresponding to its certificate. This message specifies the following:

- Signature algorithm: The signature algorithm used; in this example, it is rsa_pss_rsae_sha256.
- Signature: The signature over the entire handshake using the private key corresponding to the public key in the Certificate message

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.162 EDT|
CertificateVerify.java:1181|Consuming CertificateVerify handshake message (
"CertificateVerify": {
  "signature algorithm": rsa_pss_rsae_sha256
  "signature": {
    0000: A1 33 EE C1 54 04 8C 69   36 43 83 A4 EE 8C 5E D4  .3..T..i6C....^.
    0010: BA 4D 30 33 37 FB DE E9   F0 45 18 56 1F FB 17 03  .M037....E.V....
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    0020: 9B 58 EF 1A 69 5B 53 39   86 8C DE 9E F1 1B 7F 87  .X..i[S9........
...
```

Read Server's Finished Message

The server sends a Finished message. This message contains a Message Authentication Code (MAC) over the entire handshake.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.168 EDT|Finished.java:934|
Consuming server Finished handshake message (
"Finished": {
  "verify data": {
    0000: 2E 67 17 0E 3D 46 04 0D   9D DA 54 3E C9 2D E9 74  .g..=F....T>.-.t
    0010: 8D EE 10 49 F7 CF 67 EC   EB 11 BE 00 73 5C 5E E5  ...I..g.....s\^.
    0020: 96 5B DC AB 1F 61 DD 5F   2E 13 33 14 D4 42 52 EF  .[...a._..3..BR.
  }
}
)
...
```

Send Certificate Message

The client sends a Certificate message because the server requested client authentication through a CertificateRequest message. The certificate message specifies similar information as the server's Certificate message. The client needs to send credentials back to the sever, so its X509KeyManager is consulted. The client looks for a match between the list of accepted issuers and the certificates that are in the KeyStore. In this case, there is a match: the client has the credentials for "duke". It's now up to the server's X509TrustManager to decide whether to accept these credentials.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.190 EDT|
CertificateMessage.java:1091|Produced client Certificate message (
"Certificate": {
  "certificate_request_context": "",
  "certificate_list": [
  {
    "certificate" : {
      "version"            : "v3",
      "serial number"      : "06:ca:dd:d7:ea:fc:49:64",
      "signature algorithm": "SHA256withRSA",
      "issuer"             : "CN=Duke, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
      "not before"         : "2026-06-30 11:33:35.000 EDT",
      "not  after"         : "2036-06-27 11:33:35.000 EDT",
      "subject"            : "CN=Duke, OU=JSSE Samples, O=Demo, L=Toronto,
ST=ON, C=CA",
      "subject public key" : "RSA",
      "extensions"         : [
        {
          ObjectId: 2.5.29.37 Criticality=false
          ExtendedKeyUsages [
            clientAuth
          ]
        },
        {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
          ObjectId: 2.5.29.14 Criticality=false
          SubjectKeyIdentifier [
          KeyIdentifier [
          0000: 96 F2 A5 C0 D0 D4 EB 29   3E 85 9C 1F F4 79 CB
E2  .......)>....y..
          0010: 8A 45 05 D1                                        .E..
          ]
          ]
        }
      ]}
    "extensions": {
      <no extension>
    }
  },
]
}
)
...
```

Send CertificateVerify Message

As with the CertificateVerify message sent by the server, the certificate sent by the client is verified by the CertificateVerify message. The message is used to provide explicit proof that the client has the private key corresponding to its certificate.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.210 EDT|
CertificateVerify.java:1138|Produced client CertificateVerify handshake
message (
"CertificateVerify": {
  "signature algorithm": rsa_pss_rsae_sha256
  "signature": {
    0000: 99 3A AF 0A 75 A9 56 FE   C2 E0 49 18 4C 11 38 C1  .:..u.V...I.L.8.
    0010: 8E CD 66 76 C6 63 83 A6   AA AD 92 6C 21 08 5E 47  ..fv.c.....l!.^G
    0020: 65 C5 EA 89 27 BB 5B C7   14 79 57 8E F3 39 DA 80  e...'.[..yW..9..
...
```

Send Finished Message

The client then sends its Finished message to the sever:

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.212 EDT|Finished.java:685|
Produced client Finished handshake message (
"Finished": {
  "verify data": {
    0000: CD EB FB 84 B9 41 6C 82   EC 26 CE 7D EE E4 08 6D  .....Al..&.....m
    0010: CA 46 A4 AC 6C 0D 64 F1   DC A6 D9 63 AD 20 B1 A4  .F..l.d....c. ..
    0020: 83 52 E4 56 89 E7 31 9D   26 18 5A BF 4B 52 51 14  .R.V..1.&.Z.KRQ.
  }
}
)
...
```

The client and server have verified the Finished messages that they have received from their peers. Both sides may now send and receive application data over the connection.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Exchange Application Data, Client Sends GET Command

The server and client are ready to exchange application data. The client sends a "GET / index.html HTTP1.1" command. Note that data over the wire is encrypted:

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.223 EDT|
SSLSocketOutputRecord.java:340|WRITE: TLSv1.3 application_data, length = 28
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.223 EDT|SSLCipher.java:2041|
Plaintext before ENCRYPTION (
  0000: 47 45 54 20 2F 69 6E 64   65 78 2E 68 74 6D 6C 20  GET /index.html
  0010: 48 54 54 50 2F 31 2E 31   0D 0A 0D 0A 17 00 00 00  HTTP/1.1........
  0020: 00 00 00 00 00 00 00 00   00 00 00 00 00           .............
)
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.224 EDT|
SSLSocketOutputRecord.java:356|Raw write (
  0000: 17 03 03 00 3D 5D B7 A0   14 D7 F7 79 DE 2C 12 A0  ....=].....y.,..
  0010: 2C 28 92 79 23 09 80 FC   15 1F D2 CB 79 E3 9E 4E  ,(.y#.......y..N
  0020: 91 2F 0A 4F D6 84 DF 9C   17 12 F1 27 C9 CA 1F EC  ./.O.......'....
  0030: 58 48 AF 65 A1 31 51 97   71 DF 5F 2C 70 1D 34 47  XH.e.1Q.q._,p.4G
  0040: 7E E5                                              ..
)
...
```

Read NewSessionTicket Message

After the server receives the client's Finished message, it can send a NewSessionTicket message anytime, which contains a PSK ticket that the client can use for speeding up future handshakes.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.230 EDT|
NewSessionTicket.java:591|Consuming NewSessionTicket message (
"NewSessionTicket": {
  "ticket_lifetime"      : "86,400",
  "ticket_age_add"       : "<omitted>",
  "ticket_nonce"         : "01",
  "ticket"               : {
    0000: 57 ED EA 0C C1 30 C8 2A   72 06 14 EB F2 C9 2F 8A  W....0.*r...../.
    0010: 01 70 83 5E B3 41 1F 91   77 6A 11 59 0C 13 73 FC  .p.^.A..wj.Y..s.
    0020: 77 74 D8 FE F2 A4 A4 47   74 CF 11 90 C7 5C 8F 53  wt.....Gt....\.S
...
```

A duplicate SSLSession is created with the newly generated PSK information attached.

```
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:18.231 EDT|
SSLSessionImpl.java:260|Session initialized:  Session(1782843678073|
TLS_AES_256_GCM_SHA384)
...
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Exchange Application Data, Server Sends HTTPS Header and Data

The client receives application data from the server, first the HTTPS header, then the actual data.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.232 EDT|
SSLSocketInputRecord.java:214|READ: TLSv1.2 application_data, length = 99
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.232 EDT|
SSLSocketInputRecord.java:501|Raw read (
  0000: 06 F5 7C 22 40 10 63 9A   E2 70 EF 81 A5 DC CF E9  ..."@.c..p......
  0010: 19 A1 9C 70 62 C7 96 30   E8 67 A0 AB EA 9A 5C 58  ...pb..0.g....\X
  0020: 9F FB 45 27 69 96 86 71   19 1E 01 18 C4 96 B1 68  ..E'i..q.......h
  0030: E5 8F 06 68 46 02 E9 62   45 75 95 96 F7 12 4F 5A  ...hF..bEu....OZ
  0040: FA 92 61 40 A0 1F EE C0   1A 04 E9 EC A6 2C 28 41  ..a@.........,(A
  0050: 63 A9 84 63 E9 CE 1A 78   47 49 A3 77 23 3E 7D 06  c..c...xGI.w#>..
  0060: B4 DA 4C                                           ..L
)
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.232 EDT|
SSLSocketInputRecord.java:247|READ: TLSv1.2 application_data, length = 99
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.232 EDT|SSLCipher.java:1946|
Plaintext after DECRYPTION (
  0000: 48 54 54 50 2F 31 2E 31   20 32 30 30 20 4F 4B 0D  HTTP/1.1 200 OK.
  0010: 0A 43 6F 6E 74 65 6E 74   2D 4C 65 6E 67 74 68 3A  .Content-Length:
  0020: 20 31 36 39 39 0D 0A 43   6F 6E 74 65 6E 74 2D 54   1699..Content-T
  0030: 79 70 65 3A 20 74 65 78   74 2F 68 74 6D 6C 0D 0A  ype: text/html..
  0040: 0D 0A                                              ..
)
HTTP/1.1 200 OK
Content-Length: 1699
Content-Type: text/html

javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.233 EDT|
SSLSocketInputRecord.java:501|Raw read (
  0000: 17 03 03 06 C4                                     .....
)
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.233 EDT|
SSLSocketInputRecord.java:214|READ: TLSv1.2 application_data, length = 1732
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.236 EDT|
SSLSocketInputRecord.java:501|Raw read (
  0000: 42 1F E6 A5 ED 42 A2 35   D1 AD CB 9C D0 27 C7 4E  B....B.5.....'.N
  0010: BF 9D D7 C9 43 DB 10 92   46 32 FE 0A FB BB 45 3F  ....C...F2....E?
  0020: 75 68 35 B1 F4 07 A2 38   4B 21 28 EE F3 AA C7 14  uh5....8K!(.....
...
)
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.236 EDT|
SSLSocketInputRecord.java:247|READ: TLSv1.2 application_data, length = 1732
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.237 EDT|SSLCipher.java:1946|
Plaintext after DECRYPTION (
  0000: 3C 21 44 4F 43 54 59 50   45 20 68 74 6D 6C 20 50  <!DOCTYPE html P
  0010: 55 42 4C 49 43 20 22 2D   2F 2F 57 33 43 2F 2F 44  UBLIC "-//W3C//D
  0020: 54 44 20 58 48 54 4D 4C   20 31 2E 30 20 54 72 61  TD XHTML 1.0 Tra
  0030: 6E 73 69 74 69 6F 6E 61   6C 2F 2F 45 4E 22 0A 20  nsitional//EN".
  0040: 20 20 20 22 68 74 74 70   3A 2F 2F 77 77 77 2E 77     "http://www.w
  0050: 33 2E 6F 72 67 2F 54 52   2F 78 68 74 6D 6C 31 2F  3.org/TR/xhtml1/
  0060: 44 54 44 2F 78 68 74 6D   6C 31 2D 74 72 61 6E 73  DTD/xhtml1-trans
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
  0070: 69 74 69 6F 6E 61 6C 2E   64 74 64 22 3E 0A 3C 68  itional.dtd">.<h
  0080: 74 6D 6C 20 6C 61 6E 67   3D 22 65 6E 2D 55 53 22  tml lang="en-US"
  0090: 20 78 6D 6C 6E 73 3D 22   68 74 74 70 3A 2F 2F 77   xmlns="http://w
  00A0: 77 77 2E 77 33 2E 6F 72   67 2F 31 39 39 39 2F 78  ww.w3.org/1999/x
  00B0: 68 74 6D 6C 22 20 78 6D   6C 3A 6C 61 6E 67 3D 0A  html" xml:lang=.
  00C0: 22 65 6E 2D 55 53 22 3E   0A 3C 68 65 61 64 3E 0A  "en-US">.<head>.
  00D0: 3C 74 69 74 6C 65 3E 4A   53 53 45 20 53 61 6D 70  <title>JSSE Samp
  00E0: 6C 65 20 43 6F 64 65 3C   2F 74 69 74 6C 65 3E 0A  le Code</title>.
...
)
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en-US" xmlns="http://www.w3.org/1999/xhtml" xml:lang=
"en-US">
<head>
<title>JSSE Sample Code</title>
...
```

Read Server's Alert Message

The server sends a close_notify alert, which notifies the client that it won't send anymore messages on this connection.

```
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.242 EDT|Alert.java:242|
Received alert message (
"Alert": {
  "level"      : "warning",
  "description": "close_notify"
}
)
...
```

Close the Connection

The server closes the socket and then the TLS connection.

```
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:18.247 EDT|
SSLSocketImpl.java:1141|Closing input stream
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.248 EDT|
SSLSocketImpl.java:577|duplex close of SSLSocket
javax.net.ssl|DEBUG|30|main|2026-06-30 14:21:18.248 EDT|
SSLSocketImpl.java:1780|close the SSL connection (passive)
javax.net.ssl|TRACE|30|main|2026-06-30 14:21:18.248 EDT|
SSLSocketImpl.java:1334|Closing output stream
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Compatibility Risks and Known Issues

Enhancements to JSSE may introduce compatibility problems and other known issues, which are described in this section.

TLS 1.3 Not Directly Compatible with Previous Versions

TLS 1.3 is not directly compatible with previous versions. Although TLS 1.3 can be implemented with a backward-compatibility mode, there are still several compatibility risks to consider when upgrading to TLS 1.3:

- TLS 1.3 uses a half-close policy, while TLS 1.2 and earlier use a duplex-close policy. For applications that depend on the duplex-close policy, there may be compatibility issues when upgrading to TLS 1.3.
- The signature_algorithms_cert extension requires that pre-defined signature algorithms are used for certificate authentication. In practice, however, an application may use unsupported signature algorithms.
- The DSA signature algorithm is not supported in TLS 1.3. If a server is configured to only use DSA certificates, it cannot negotiate a TLS 1.3 connection.
- The supported cipher suites for TLS 1.3 are not the same as TLS 1.2 and earlier. If an application hardcodes cipher suites that are no longer supported, it may not be able to use TLS 1.3 without modifications to its code, for example TLS_AES_128_GCM_SHA256 (1.3 and later) versus TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (1.2 and earlier).
- The TLS 1.3 session resumption and key update behaviors are different from TLS 1.2 and earlier. The compatibility impact should be minimal, but it could be a risk if an application depends on the handshake details of the TLS protocols.

Code Examples

The following code examples are included in this section:

Topics

- Converting an Unsecure Socket to a Secure Socket
- Running the JSSE Sample Code
- Creating a Keystore to Use with JSSE
- Using the Server Name Indication (SNI) Extension

Converting an Unsecure Socket to a Secure Socket

Example 8-26 shows sample code that can be used to set up communication between a client and a server using unsecure sockets. This code is then modified in Example 8-27 to use JSSE to set up secure socket communication.

Example 8-26 Socket Example Without SSL

The following examples demonstrates server-side and client-side code for setting up an unsecure socket connection.

Copyright © 1993, 2026, Oracle and/or its affiliates.

In a Java program that acts as a server and communicates with a client using sockets, the socket communication is set up with code similar to the following:

```
import java.io.*;
import java.net.*;

. . .

int port = availablePortNumber;

ServerSocket s;

try {
    s = new ServerSocket(port);
    Socket c = s.accept();

    OutputStream out = c.getOutputStream();
    InputStream in = c.getInputStream();

    // Send messages to the client through
    // the OutputStream
    // Receive messages from the client
    // through the InputStream
} catch (IOException e) { }
```

The client code to set up communication with a server using sockets is similar to the following:

```
import java.io.*;
import java.net.*;

. . .

int port = availablePortNumber;
String host = "hostname";

try {
    s = new Socket(host, port);

    OutputStream out = s.getOutputStream();
    InputStream in = s.getInputStream();

    // Send messages to the server through
    // the OutputStream
    // Receive messages from the server
    // through the InputStream
} catch (IOException e) { }
```

Example 8-27 Socket Example with SSL

The following examples demonstrate server-side and client-side code for setting up a secure socket connection.

Copyright © 1993, 2026, Oracle and/or its affiliates.

In a Java program that acts as a server and communicates with a client using secure sockets, the socket communication is set up with code similar to the following. Differences between this program and the one for communication using unsecure sockets are highlighted in bold.

```
import java.io.*;
import javax.net.ssl.*;

. . .

int port = availablePortNumber;

SSLServerSocket s;

try {
    SSLServerSocketFactory sslSrvFact =
        (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
    s = (SSLServerSocket)sslSrvFact.createServerSocket(port);

    SSLSocket c = (SSLSocket)s.accept();

    OutputStream out = c.getOutputStream();
    InputStream in = c.getInputStream();

    // Send messages to the client through
    // the OutputStream
    // Receive messages from the client
    // through the InputStream
}

catch (IOException e) {
}
```

The client code to set up communication with a server using secure sockets is similar to the following, where differences with the unsecure version are highlighted in bold:

```
import java.io.*;
import javax.net.ssl.*;

. . .

int port = availablePortNumber;
String host = "hostname";

try {
    SSLSocketFactory sslFact =
        (SSLSocketFactory)SSLSocketFactory.getDefault();
    SSLSocket s = (SSLSocket)sslFact.createSocket(host, port);

    OutputStream out = s.getOutputStream();
    InputStream in = s.getInputStream();

    // Send messages to the server through
    // the OutputStream
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        // Receive messages from the server
        // through the InputStream
    }

    catch (IOException e) {
    }
```

Running the JSSE Sample Code

The JSSE sample programs illustrate how to use JSSE.

> **Note**
>
> When you use the sample code, be aware that the sample programs are designed to illustrate how to use JSSE. They are not designed to be robust applications.
>
> Setting up secure communications involves complex algorithms. The sample programs provide no feedback during the setup process. When you run the programs, be patient: you may not see any output for a while. If you run the programs with the `javax.net.debug` system property set to `all`, you will see more feedback. For an introduction to reading this debug information, see Debugging TLS Connections.

This section contains the following topics:

- Where to Find the Sample Code
- Sample Certificates and Keys
- The following topics describe the samples:
  - Sample Code Illustrating a Secure Socket Connection Between a Client and a Server
  - Sample Code Illustrating HTTPS Connections
  - Sample Code Illustrating a Secure RMI Connection
  - Sample Code Illustrating the Use of an SSLEngine
- Troubleshooting JSSE Sample Code

Where to Find the Sample Code

JSSE Sample Code in the Java SE 8 documentation lists all the sample code files and text files. That page also provides a link to a ZIP file that you can download to obtain all the sample code files.

Sample Certificates and Keys

The JSSE samples use the following certificate keystore files to authenticate the clients and servers:

- `*/testkeys`

  These files are used by the code samples as the source of public/private key and certificate material. In the client program directories, the `testkeys` files contains the certificate entry for the Java mascot `Duke`. In the server program directories (`./sockets/server` and `rmi`), the file contains a certificate entry for the server `localhost`.

  The sample code expects the `testkeys` file to be in the current working directory.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> These are very simple certificates and are not appropriate for a production environment, but they should be sufficient for running the samples here.
>
> The password for these keystores is: `passphrase`

- `samplecacerts`

  This truststore file is very similar to the stock JDK `cacerts` file, in that it contains trust certificates from several vendors. It also contains the trusted certificates from `Duke` and `localhost`.

  The password for this keystore is `changeit`

  Please see your provider's documentation for how to configure the location of your trusted certificate file.

  > **Note**
  >
  > Users of the JDK can specify the location of the truststore by using one of the following methods:

  1. System properties:

     ```
     java -Djavax.net.ssl.trustStore=samplecacerts \
          -Djavax.net.ssl.trustStorePassword=changeit Application
     ```

  2. Install the file into:

     ```
     <java-home>/lib/security/jssecacerts
     ```

  3. Install the file into:

     ```
     <java-home>/lib/security/cacerts
     ```

  If you choose (2) or (3), be sure to replace this file with a production `cacerts` file before deployment.

The utility `keytool` can be used to generate alternate certificates and keystore files.

> **Note**
>
> Ensure that you verify your `cacerts` file. Since you trust the CAs in the `cacerts` file as entities for signing and issuing certificates to other entities, you must manage the `cacerts` file carefully. The `cacerts` file should contain only certificates of the entities and CAs you trust. It is your responsibility to verify the trusted root CA certificates bundled in the `cacerts` file and make your own trust decisions. To remove an untrusted CA certificate from the `cacerts` file, use the `-delete` command of the `keytool` utility with the `-cacerts` option. Contact your system administrator if you do not have permission to edit this file.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Alternatively, you can use your own truststore and keystore files. See Creating a Keystore to Use with JSSE.

Sample Code Illustrating a Secure Socket Connection Between a Client and a Server

The sample programs in the `samples/sockets` directory illustrate how to set up a secure socket connection between a client and a server.

When running the sample client programs, you can communicate with an existing server, such as a web server, or you can communicate with the sample server program, `ClassFileServer`. You can run the sample client and the sample server programs on different machines connected to the same network, or you can run them both on one machine but from different terminal windows.

All the sample `SSLSocketClient*` programs in the samples/sockets/client directory (and `URLReader*` programs described in Sample Code Illustrating HTTPS Connections) can be run with the `ClassFileServer` sample server program. An example of how to do this is shown in Running SSLSocketClientWithClientAuth with ClassFileServer. You can make similar changes to run `URLReader`, `SSLSocketClient`, or `SSLSocketClientWithTunneling` with `ClassFileServer`.

If an authentication error occurs during communication between the client and the server (whether using a web server or `ClassFileServer`), it is most likely because the necessary keys are not in the truststore (trust key database). See Terms and Definitions. For example, the `ClassFileServer` uses a keystore called `testkeys` containing the private key for `localhost` as needed during the SSL handshake. The `testkeys` keystore is included in the same `samples/sockets/server` directory as the `ClassFileServer` source. If the client cannot find a certificate for the corresponding public key of `localhost` in the truststore it consults, then an authentication error will occur. Be sure to use the `samplecacerts` truststore (which contains the public key and certificate of the `localhost`), as described in the next section.

Configuration Requirements

When running the sample programs that create a secure socket connection between a client and a server, you will need to make the appropriate certificates file (truststore) available. For both the client and the server programs, you should use the certificates file `samplecacerts` from the `samples` directory. Using this certificates file will allow the client to authenticate the server. The file contains all the common Certificate Authority (CA) certificates shipped with the JDK (in the cacerts file), plus a certificate for `localhost` needed by the client to authenticate `localhost` when communicating with the sample server `ClassFileServer`. The `ClassFileServer` uses a keystore containing the private key for `localhost` that corresponds to the public key in `samplecacerts`.

To make the `samplecacerts` file available to both the client and the server, you can either copy it to the file `<java-home>/lib/security/jssecacerts`, rename it to cacerts, and use it to replace the `<java-home>/lib/security/cacerts` file, or add the following option to the command line when running the `java` command for both the client and the server:

```
-Djavax.net.ssl.trustStore=path_to_samplecacerts_file
```

To know more about <*java-home*>, see Terms and Definitions.

The password for the `samplecacerts` truststore is `changeit`. You can substitute your own certificates in the samples by using the `keytool` utility.

If you use a browser, such as Mozilla Firefox or Microsoft Internet Explorer, to access the sample SSL server provided in the `ClassFileServer` example, then a dialog box may pop up

Copyright © 1993, 2026, Oracle and/or its affiliates.

with the message that it does not recognize the certificate. This is normal because the certificate used with the sample programs is self-signed and is for testing only. You can accept the certificate for the current session. After testing the SSL server, you should exit the browser, which deletes the test certificate from the browser's namespace.

For client authentication, a separate `duke` certificate is available in the appropriate directories. The public key and certificate is also stored in the `samplecacerts` file.

Running SSLSocketClient

The `SSLSocketClient.java` program in JSSE Sample Code in the Java SE 8 documentation demonstrates how to create a client that uses an `SSLSocket` to send an HTTP request and to get a response from an HTTPS server. The output of this program is the HTML source for `https://www.verisign.com/index.html`.

You must not be behind a firewall to run this program as provided. If you run it from behind a firewall, you will get an `UnknownHostException` because JSSE cannot find a path through your firewall to `www.verisign.com`. To create an equivalent client that can run from behind a firewall, set up proxy tunneling as illustrated in the sample program `SSLSocketClientWithTunneling`.

Running SSLSocketClientWithTunneling

The `SSLSocketClientWithTunneling.java` program in JSSE Sample Code in the Java SE 8 documentation illustrates how to do proxy tunneling to access a secure web server from behind a firewall. To run this program, you must set the following Java system properties to the appropriate values:

```
java -Dhttps.proxyHost=webproxy
-Dhttps.proxyPort=ProxyPortNumber
SSLSocketClientWithTunneling
```

> **Note**
>
> Proxy specifications with the `-D` options are optional. Replace *`webproxy`* with the name of your proxy host and *`ProxyPortNumber`* with the appropriate port number.

The program will return the HTML source file from `https://www.verisign.com/index.html`.

Running SSLSocketClientWithClientAuth

The `SSLSocketClientWithClientAuth.java` program in JSSE Sample Code in the Java SE 8 documentation shows how to set up a key manager to do client authentication if required by a server. This program also assumes that the client is not outside a firewall. You can modify the program to connect from inside a firewall by following the example in `SSLSocketClientWithTunneling`.

To run this program, you must specify three parameters: host name, port number, and requested file path. To mirror the previous examples, you can run this program without client authentication by setting the host to `www.verisign.com`, the port to `443`, and the requested file path to `https://www.verisign.com/`. The output when using these parameters is the HTML for the website `https://www.verisign.com/`.

To run `SSLSocketClientWithClientAuth` to do client authentication, you must access a server that requests client authentication. You can use the sample program `ClassFileServer` as this server. This is described in the following sections.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Running ClassFileServer

The program referred to herein as `ClassFileServer` is made up of two files: `ClassFileServer.java` and `ClassServer.java` in JSSE Sample Code in the Java SE 8 documentation.

To execute them, run `ClassFileServer.class`, which requires the following parameters:

- `port` can be any available unused port number, for example, you can use the number `2001`.
- `docroot` indicates the directory on the server that contains the file you want to retrieve. For example, on Linux, you can use `/home/`*`userid`*`/` (where *`userid`* refers to your particular UID), whereas on Windows, you can use `c:\`.
- `TLS` is an optional parameter that indicates that the server is to use SSL or TLS.
- `true` is an optional parameter that indicates that client authentication is required. This parameter is only consulted if the TLS parameter is set.

> **Note**
>
> The `TLS` and `true` parameters are optional. If you omit them, indicating that an ordinary (not TLS) file server should be used, without authentication, then nothing happens. This is because one side (the client) is trying to negotiate with TLS, while the other (the server) is not, so they cannot communicate.
>
> The server expects GET requests in the form `GET /path_to_file`.

Running SSLSocketClientWithClientAuth with ClassFileServer

You can use the sample programs `SSLSocketClientWithClientAuth.java` and `ClassFileServer` in JSSE Sample Code in the Java SE 8 documentation to set up authenticated communication, where the client and server are authenticated to each other. You can run both sample programs on different machines connected to the same network, or you can run them both on one machine but from different terminal windows or command prompt windows. To set up both the client and the server, do the following:

1. Run the program `ClassFileServer` from one machine or terminal window. See Running ClassFileServer.
2. Run the program `SSLSocketClientWithClientAuth` on another machine or terminal window. `SSLSocketClientWithClientAuth` requires the following parameters:
   - `host` is the host name of the machine that you are using to run `ClassFileServer`.
   - `port` is the same port that you specified for `ClassFileServer`.
   - `requestedfilepath` indicates the path to the file that you want to retrieve from the server. You must give this parameter as `/filepath`. Forward slashes are required in the file path because it is used as part of a GET statement, which requires forward slashes regardless of what type of operating system you are running. The statement is formed as follows:

     ```
     "GET " + requestedfilepath + " HTTP/1.0"
     ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> You can modify the other `SSLClient*` applications' `GET` commands to connect to a local machine running `ClassFileServer`.

Sample Code Illustrating HTTPS Connections

There are two primary APIs for accessing secure communications through JSSE. One way is through a socket-level API that can be used for arbitrary secure communications, as illustrated by the `SSLSocketClient`, `SSLSocketClientWithTunneling`, and `SSLSocketClientWithClientAuth` (with and without `ClassFileServer`) sample programs.

A second, and often simpler, way is through the standard Java URL API. You can communicate securely with an SSL-enabled web server by using the HTTPS URL protocol or scheme using the `java.net.URL` class.

Support for HTTPS URL schemes is implemented in many of the common browsers, which allows access to secured communications without requiring the socket-level API provided with JSSE.

An example URL is `https://www.verisign.com`.

The trust and key management for the HTTPS URL implementation is environment-specific. The JSSE implementation provides an HTTPS URL implementation. To use a different HTTPS protocol implementation, set the `java.protocol.handler.pkgs`. See How to Specify a java.lang.System Property to the package name. See the `java.net.URL` class documentation for details.

The samples that you can download with JSSE include two sample programs that illustrate how to create an HTTPS connection. Both of these sample programs (`URLReader.java` and `URLReaderWithOptions.java`) are in the `samples/urls` directory.

Running URLReader

The `URLReader.java` program in JSSE Sample Code in the Java SE 8 documentation illustrates using the URL class to access a secure site. The output of this program is the HTML source for `https://www.verisign.com/`. By default, the HTTPS protocol implementation included with JSSE is used. To use a different implementation, set the system property `java.protocol.handler.pkgs` value to be the name of the package containing the implementation.

If you are running the sample code behind a firewall, then you must set the `https.proxyHost` and `https.proxyPort` system properties. For example, to use the proxy host "webproxy" on port 8080, you can use the following options for the `java` command:

```
-Dhttps.proxyHost=webproxy
-Dhttps.proxyPort=8080
```

Alternatively, you can set the system properties within the source code with the `java.lang.System` method `setProperty()`. For example, instead of using the command-line options, you can include the following lines in your program:

```
System.setProperty("java.protocol.handler.pkgs", "com.ABC.myhttpsprotocol");
System.setProperty("https.proxyHost", "webproxy");
System.setProperty("https.proxyPort", "8080");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Running URLReaderWithOptions

The `URLReaderWithOptions.java` program in JSSE Sample Code in the Java SE 8 documentation is essentially the same as the `URLReader.java` program, except that it allows you to optionally input any or all of the following system properties as arguments to the program when you run it:

- `java.protocol.handler.pkgs`
- `https.proxyHost`
- `https.proxyPort`
- `https.cipherSuites`

To run `URLReaderWithOptions`, enter the following command:

```
java URLReaderWithOptions [-h proxyhost -p proxyport] [-k
protocolhandlerpkgs] [-c ciphersarray]
```

> **Note**
>
> Multiple protocol handlers can be included in the `protocolhandlerpkgs` argument as a list with items separated by vertical bars. Multiple SSL cipher suite names can be included in the `ciphersarray` argument as a list with items separated by commas. The possible cipher suite names are the same as those returned by the `SSLSocket.getSupportedCipherSuites()` method. The suite names are taken from the SSL and TLS protocol specifications.

You need a `protocolhandlerpkgs` argument only if you want to use an HTTPS protocol handler implementation other than the default one provided by Oracle.

If you are running the sample code behind a firewall, then you must include arguments for the proxy host and the proxy port. Additionally, you can include a list of cipher suites to enable.

Here is an example of running `URLReaderWithOptions` and specifying the proxy host "webproxy" on port 8080:

```
java URLReaderWithOptions -h webproxy -p 8080
```

Sample Code Illustrating a Secure RMI Connection

The sample code in the `samples/rmi` directory illustrates how to create a secure Java Remote Method Invocation (RMI) connection. The sample code is basically a "Hello World" example modified to install and use a custom RMI socket factory.

Sample Code Illustrating the Use of an SSLEngine

`SSLEngine` gives application developers flexibility when choosing I/O and compute strategies. Rather than tie the SSL/TLS implementation to a specific I/O abstraction (such as single-threaded `SSLSockets`), `SSLEngine` removes the I/O and compute constraints from the SSL/TLS implementation.

As mentioned earlier, `SSLEngine` is an advanced API, and is not appropriate for casual use. Some introductory sample code is provided here that helps illustrate its use. The first demo removes most of the I/O and threading issues, and focuses on many of the SSLEngine

Copyright © 1993, 2026, Oracle and/or its affiliates.

methods. The second demo is a more realistic example showing how `SSLEngine` might be combined with Java NIO to create a rudimentary HTTP/HTTPS server.

Running SSLEngineSimpleDemo

The `SSLEngineSimpleDemo.java` program in JSSE Sample Code in the Java SE 8 documentation is a very simple application that focuses on the operation of the `SSLEngine` while simplifying the I/O and threading issues. This application creates two `SSLEngine` objects that exchange SSL/TLS messages via common `ByteBuffer` objects. A single loop serially performs all of the engine operations and demonstrates how a secure connection is established (handshaking), how application data is transferred, and how the engine is closed.

The `SSLEngineResult` provides a great deal of information about the current state of the `SSLEngine`. This example does not examine all of the states. It simplifies the I/O and threading issues to the point that this is not a good example for a production environment; nonetheless, it is useful to demonstrate the overall function of the `SSLEngine`.

Troubleshooting JSSE Sample Code

One of the most common problems people have in using JSSE is when the JSSE receives a certificate that is unknown to the mechanism that makes trust decisions. If an unknown certificate is received, the trust mechanism will throw an exception saying that the certificate is untrusted. Make sure that the correct trust store is being used, and that the JSSE is installed and configured correctly.

If you are using the "localhost" or "duke" credentials, be sure that you have correctly specified the location of the samplecacerts file, otherwise your application will not work. (See Sample Certificates and Keys for more information.)

The SSL debug mechanism can be used to investigate such trust problems. See the implementation documentation for more information about this subject.

Creating a Keystore to Use with JSSE

This section demonstrates how you can use the `keytool` utility to create a simple PKCS12 keystore suitable for use with JSSE.

First you make a `keyEntry` (with public and private keys) in the keystore, and then you make a corresponding `trustedCertEntry` (public keys only) in a truststore. For client authentication, you follow a similar process for the client's certificates.

> **Note**
>
> It is beyond the scope of this example to explain each step in detail. See The keytool Command in *Java Development Kit Tool Specifications* for more information.

User input is shown in bold.

1. Create a new keystore and self-signed certificate with corresponding public and private keys.

```
    % keytool -genkeypair -alias duke -keyalg RSA -validity 7 -keystore
keystore

    Enter keystore password:  <password>
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
What is your first and last name?
[Unknown]:  Duke
What is the name of your organizational unit?
[Unknown]:  Java Software
What is the name of your organization?
[Unknown]:  Oracle, Inc.
What is the name of your City or Locality?
[Unknown]:  Palo Alto
What is the name of your State or Province?
[Unknown]:  CA
What is the two-letter country code for this unit?
[Unknown]:  US
Is CN=Duke, OU=Java Software, O="Oracle, Inc.",
L=Palo Alto, ST=CA, C=US correct?
[no]:  yes
```

2. Examine the keystore. Notice that the entry type is `PrivatekeyEntry`, which means that this entry has a private key associated with it).

```
% keytool -list -v -keystore keystore

Enter keystore password:  <password>

Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: duke
Creation date: Jul 25, 2016
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto,
ST=CA, C=US
Issuer: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto,
ST=CA, C=US
Serial number: 210cccfc
Valid from: Mon Jul 25 10:33:27 IST 2016 until: Mon Aug 01 10:33:27
IST 2016
Certificate fingerprints:
     SHA1: 80:E5:8A:47:7E:4F:5A:70:83:97:DD:F4:DA:29:3D:15:6B:2A:45:1F
     SHA256:
ED:3C:70:68:4E:86:35:9C:63:CC:B9:59:35:58:94:1F:7E:B8:B0:EE:D2:
4B:9D:80:31:67:8A:D4:B4:7A:B5:12
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: RSA (2048)
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 7F C9 95 48 42 8D 68 91   BA 1E E6 5C 2C 6B FF
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
75  ...HB.h....\,k.u
   0010: 5F 19 78 43                                        _.xC
   ]
   ]
```

3. Export and examine the self-signed certificate.

```
% keytool -export -alias duke -keystore keystore -rfc -file duke.cer
Enter keystore password:  <password>
Certificate stored in file <duke.cer>
% cat duke.cer
-----BEGIN CERTIFICATE-----
MIIDdzCCAl+gAwIBAgIEIQzM/DANBgkqhkiG9w0BAQsFADBsMQswCQYDVQQGEwJV
UzELMAkGA1UECBMCQ0ExEjAQBgNVBAcTCVBhbG8gQWx0bzEVMBMGA1UEChMMT3Jh
Y2xlLCBJbmMuMRYwFAYDVQQLEw1KYXZhIFNvZnR3YXJlMQ0wCwYDVQQDEwREdWtl
MB4XDTE2MDcyNTA1MDMyN1oXDTE2MDgwMTA1MDMyN1owbDELMAkGA1UEBhMCVVMx
CzAJBgNVBAgTAkNBMRIwEAYDVQQHEwlQYWxvIEFsdG8xFTATBgNVBAoTDE9yYWNs
ZSwgSW5jLjEWMBQGA1UECxMNSmF2YSBTb2Z0d2FyZTENMAsGA1UEAxMERHVrZTCC
ASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAJ7+Yeu6HDZgWwkGlG4iKH9w
vGKrxXVR57FaFyheMevrgj1ovVnQVFhfdMvjPkjWmpqLg6rfTqU4bKbtoMWV6+Rn
uQrCw2w9xNC93hX9PxRa20UKrSRDKnUSvi1wjlaxfj0KUKuMwbbY9S8x/naYGeTL
lwbHiiMvkoFkP2kzhVgeqHjIwSz4HRN8vWHCwgIDFWX/ZlS+LbvB4TSZkS0ZcQUV
vJWTocOd8RB90W3bkibWkWq166XYGE1Nq1L4WIhrVJwbav6ual69yJsEpVcshVkx
E1WKzJg7dGb03to4agbReb6+aoCUwb2vNUudNWasSrxoEFArVFGD/ZkPT0esfqEC
AwEAAaMhMB8wHQYDVR0OBBYEFH/JlUhCjWiRuh7mXCxr/3VfGXhDMA0GCSqGSIb3
DQEBCwUAA4IBAQAmcTm2ahsIJLayajsvm8yPzQsHA7kIwWfPPHCoHmNbynG67oHB
fleaNvrgm/raTT3TrqQkg0525qI6Cqaoyy8JA2fAp3i+hmyoGHaIlo14bKazaiPS
RCCqk0J8vwY3CY9nVal1XlHJMEcYV7X1sxKbuAKFoAJ29E/p6ie0JdHtQe31M7X9
FNLYzt8EpJYUtWo13B9Oufz/Guuex9PQ7aC93rbO32MxtnnCGMxQHlaHLLPygc/x
cffGz5Xe5s+NEm78CY7thgN+drI7icBYmv4navsnr2OQaD3AfnJ4WYSQyyUUCPxN
zuk+B0fbLn7PCCcQspmqfgzIpgbEM9M1/yav
-----END CERTIFICATE-----
```

Alternatively, you could generate a Certificate Signing Request (CSR) with the `-certreq` command and send that to a Certificate Authority (CA) for signing. See Requesting a Signed Certificate from a CA in The keytool Command for an example.

4. Import the certificate into a new truststore.

```
% keytool -import -alias dukecert -file duke.cer -keystore truststore
Enter keystore password:  <password>
Re-enter new password:
Owner: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto,
ST=CA, C=US
Issuer: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto,
ST=CA, C=US
Serial number: 210cccfc
Valid from: Mon Jul 25 10:33:27 IST 2016 until: Mon Aug 01 10:33:27
IST 2016
Certificate fingerprints:
     SHA1: 80:E5:8A:47:7E:4F:5A:70:83:97:DD:F4:DA:29:3D:15:6B:2A:45:1F
     SHA256:
ED:3C:70:68:4E:86:35:9C:63:CC:B9:59:35:58:94:1F:7E:B8:B0:EE:D2:
4B:9D:80:31:67:8A:D4:B4:7A:B5:12
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    Signature algorithm name: SHA256withRSA
    Subject Public Key Algorithm: RSA (2048)
    Version: 3

    Extensions:

    #1: ObjectId: 2.5.29.14 Criticality=false
    SubjectKeyIdentifier [
    KeyIdentifier [
    0000: 7F C9 95 48 42 8D 68 91   BA 1E E6 5C 2C 6B FF
75  ...HB.h....\,k.u
    0010: 5F 19 78 43                                        _.xC
    ]
    ]

    Trust this certificate? [no]:  yes
    Certificate was added to keystore
```

5. Examine the truststore. Note that the entry type is `trustedCertEntry`, which means that a private key is not available for this entry. It also means that this file is not suitable as a keystore of the `KeyManager`.

```
    % keytool -list -v -keystore truststore
    Enter keystore password:  <password>

    Keystore type: PKCS12
    Keystore provider: SUN

    Your keystore contains 1 entry

    Alias name: dukecert
    Creation date: Jul 25, 2016
    Entry type: trustedCertEntry

    Owner: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto,
ST=CA, C=US
    Issuer: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto,
ST=CA, C=US
    Serial number: 210cccfc
    Valid from: Mon Jul 25 10:33:27 IST 2016 until: Mon Aug 01 10:33:27
IST 2016
    Certificate fingerprints:
         SHA1: 80:E5:8A:47:7E:4F:5A:70:83:97:DD:F4:DA:29:3D:15:6B:2A:45:1F
         SHA256:
ED:3C:70:68:4E:86:35:9C:63:CC:B9:59:35:58:94:1F:7E:B8:B0:EE:D2:
    4B:9D:80:31:67:8A:D4:B4:7A:B5:12
    Signature algorithm name: SHA256withRSA
    Subject Public Key Algorithm: RSA (2048)
    Version: 3

    Extensions:

    #1: ObjectId: 2.5.29.14 Criticality=false
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    SubjectKeyIdentifier [
    KeyIdentifier [
    0000: 7F C9 95 48 42 8D 68 91   BA 1E E6 5C 2C 6B FF
75  ...HB.h....\,k.u
    0010: 5F 19 78 43                                        _.xC
    ]
    ]



    ****************************************
    ****************************************
```

6. Now run your applications with the appropriate keystores. Because this example assumes that the default `X509KeyManager` and `X509TrustManager` are used, you select the keystores using the system properties described in Customizing JSSE.

```
% java -Djavax.net.ssl.keyStore=keystore -
Djavax.net.ssl.keyStorePassword=password Server
% java -Djavax.net.ssl.trustStore=truststore -
Djavax.net.ssl.trustStorePassword=trustword Client
```

> **Note**
>
> This example authenticated the server only. For client authentication, provide a similar keystore for the client's keys and an appropriate truststore for the server.

Using the Server Name Indication (SNI) Extension

These examples illustrate how you can use the Server Name Indication (SNI) Extension for client-side and server-side applications, and how it can be applied to a virtual infrastructure.

For all examples in this section, to apply the parameters after you set them, call the `setSSLParameters(SSLParameters)` method on the corresponding `SSLSocket`, `SSLEngine`, or `SSLServerSocket` object.

Typical Client-Side Usage Examples

The following is a list of use cases that require understanding of the SNI extension for developing a client application:

- Case 1. The client wants to access `www.example.com`.

  Set the host name explicitly:

  ```
  SNIHostName serverName = new SNIHostName("www.example.com");
  sslParameters.setServerNames(Collections.singletonList(serverName));
  ```

  The client should always specify the host name explicitly.
- Case 2. The client does not want to use SNI because the server does not support it.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Disable SNI with an empty server name list:

```
sslParameters.setServerNames(Collections.emptyList());
```

- Case 3. The client wants to access URL `https://www.example.com`.

  Oracle providers will set the host name in the SNI extension by default, but third-party providers may not support the default server name indication. To keep your application provider-independent, always set the host name explicitly.

- Case 4. The client wants to switch a socket from server mode to client mode.

  First switch the mode with the following method: `sslSocket.setUseClientMode(true)`. Then reset the server name indication parameters on the socket.

Typical Server-Side Usage Examples

The following is a list of use cases that require understanding of the SNI extension for developing a server application:

- Case 1. The server wants to accept all server name indication types.

  If you do not have any code dealing with the SNI extension, then the server ignores all server name indication types.

- Case 2. The server wants to deny all server name indications of type `host_name`.

  Set an invalid server name pattern for `host_name`:

  ```
  SNIMatcher matcher = SNIHostName.createSNIMatcher("");
  Collection<SNIMatcher> matchers = new ArrayList<>(1);
  matchers.add(matcher);
  sslParameters.setSNIMatchers(matchers);
  ```

  Another way is to create an `SNIMatcher` subclass with a `matches()` method that always returns `false`:

  ```
  class DenialSNIMatcher extends SNIMatcher {
      DenialSNIMatcher() {
          super(StandardConstants.SNI_HOST_NAME);
      }

      @Override
      public boolean matches(SNIServerName serverName) {
          return false;
      }
  }

  SNIMatcher matcher = new DenialSNIMatcher();
  Collection<SNIMatcher> matchers = new ArrayList<>(1);
  matchers.add(matcher);
  sslParameters.setSNIMatchers(matchers);
  ```

- Case 3. The server wants to accept connections to any host names in the `example.com` domain.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Set the recognizable server name for `host_name` as a pattern that includes all `*.example.com` addresses:

```
    SNIMatcher matcher = SNIHostName.createSNIMatcher("(.*\\.)*example\
\.com");
    Collection<SNIMatcher> matchers = new ArrayList<>(1);
    matchers.add(matcher);
    sslParameters.setSNIMatchers(matchers);
```

- Case 4. The server wants to switch a socket from client mode to server mode.

  First switch the mode with the following method: `sslSocket.setUseClientMode(false)`. Then reset the server name indication parameters on the socket.

Working with Virtual Infrastructures

This section describes how to use the Server Name Indication (SNI) extension from within a virtual infrastructure. It illustrates how to create a parser for ClientHello messages from a socket, provides examples of virtual server dispatchers using `SSLSocket` and `SSLEngine`, describes what happens when the SNI extension is not available, and demonstrates how to create a failover `SSLContext`.

Preparing the ClientHello Parser

Applications must implement an API to parse the ClientHello messages from a socket. The following examples illustrate the `SSLCapabilities` and `SSLExplorer` classes that can perform these functions.

SSLSocketClient.java encapsulates the TLS/DTLS security capabilities during handshaking (that is, the list of cipher suites to be accepted in an TLS/DTLS handshake, the record version, the hello version, and the server name indication). It can be retrieved by exploring the network data of an TLS/DTLS connection via the `SSLExplorer.explore()` method.

SSLExplorer.java explores the initial ClientHello message from a TLS client, but it does not initiate handshaking or consume network data. The `SSLExplorer.explore()` method parses the ClientHello message, and retrieves the security parameters into `SSLCapabilities`. The method must be called before handshaking occurs on any TLS connections.

Virtual Server Dispatcher Based on SSLSocket

This section describes the procedure for using a virtual server dispatcher based on `SSLSocket`.

1. **Register the server name handler.**

   At this step, the application may create different `SSLContext` objects for different server name indications, or link a certain server name indication to a specified virtual machine or distributed system.

   For example, if the server name is `www.example.org`, then the registered server name handler may be for a local virtual hosting web service. The local virtual hosting web service will use the specified `SSLContext`. If the server name is `www.example.com`, then the registered server name handler may be for a virtual machine hosting on `10.0.0.36`. The handler may map this connection to the virtual machine.

2. **Create a `ServerSocket` and accept the new connection.**

   ```
   ServerSocket serverSocket = new ServerSocket(serverPort);
   Socket socket = serverSocket.accept();
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

3. **Read and buffer bytes from the socket input stream, and then explore the buffered bytes.**

```
InputStream ins = socket.getInputStream();
byte[] buffer = new byte[0xFF];
int position = 0;
SSLCapabilities capabilities = null;

// Read the header of TLS record
while (position < SSLExplorer.RECORD_HEADER_SIZE) {
    int count = SSLExplorer.RECORD_HEADER_SIZE - position;
    int n = ins.read(buffer, position, count);
    if (n < 0) {
        throw new Exception("unexpected end of stream!");
    }
    position += n;
}

// Get the required size to explore the SSL capabilities
int recordLength = SSLExplorer.getRequiredSize(buffer, 0, position);
if (buffer.length < recordLength) {
    buffer = Arrays.copyOf(buffer, recordLength);
}

while (position < recordLength) {
    int count = recordLength - position;
    int n = ins.read(buffer, position, count);
    if (n < 0) {
        throw new Exception("unexpected end of stream!");
    }
    position += n;
}

// Explore
capabilities = SSLExplorer.explore(buffer, 0, recordLength);
if (capabilities != null) {
    System.out.println("Record version: " +
capabilities.getRecordVersion());
    System.out.println("Hello version: " + capabilities.getHelloVersion());
}
```

4. **Get the requested server name from the explored capabilities.**

```
List<SNIServerName> serverNames = capabilities.getServerNames();
```

5. **Look for the registered server name handler for this server name indication.**

   If the service of the host name is resident in a virtual machine or another distributed system, then the application must forward the connection to the destination. The application will need to read and write the raw internet data, rather then the SSL application from the socket stream.

```
Socket destinationSocket = new Socket(serverName, 443);
// Forward buffered bytes and network data from the current socket to the
destinationSocket.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

If the service of the host name is resident in the same process, and the host name service can use the `SSLSocket` directly, then the application will need to set the `SSLSocket` instance to the server:

```
// Get service context from registered handler
// or create the context
SSLContext serviceContext = ...

SSLSocketFactory serviceSocketFac = serviceContext.getSSLSocketFactory();

// wrap the buffered bytes
ByteArrayInputStream bais = new ByteArrayInputStream(buffer, 0, position);
SSLSocket serviceSocket = (SSLSocket)serviceSocketFac.createSocket(socket,
bais, true);

// Now the service can use serviceSocket as usual.
```

Virtual Server Dispatcher Based on SSLEngine

This section describes the procedure for using a virtual server dispatcher based on `SSLEngine`.

1. **Register the server name handler.**

   At this step, the application may create different `SSLContext` objects for different server name indications, or link a certain server name indication to a specified virtual machine or distributed system.

   For example, if the server name is `www.example.org`, then the registered server name handler may be for a local virtual hosting web service. The local virtual hosting web service will use the specified `SSLContext`. If the server name is `www.example.com`, then the registered server name handler may be for a virtual machine hosting on `10.0.0.36`. The handler may map this connection to the virtual machine.

2. **Create a `ServerSocket` or `ServerSocketChannel` and accept the new connection.**

   ```
   ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
   serverSocketChannel.bind(...);
   ...
   SocketChannel socketChannel = serverSocketChannel.accept();
   ```

3. **Read and buffer bytes from the socket input stream, and then explore the buffered bytes.**

   ```
   ByteBuffer buffer = ByteBuffer.allocate(0xFF);
   SSLCapabilities capabilities = null;
   while (true) {
       // ensure the capacity
       if (buffer.remaining() == 0) {
           ByteBuffer oldBuffer = buffer;
           buffer = ByteBuffer.allocate(buffer.capacity() + 0xFF);
           buffer.put(oldBuffer);
       }

       int n = sc.read(buffer);
       if (n < 0) {
           throw new Exception("unexpected end of stream!");
       }
   ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            int position = buffer.position();
            buffer.flip();
            capabilities = explorer.explore(buffer);
            buffer.rewind();
            buffer.position(position);
            buffer.limit(buffer.capacity());
            if (capabilities != null) {
                System.out.println("Record version: " +
                    capabilities.getRecordVersion());
                System.out.println("Hello version: " +
                    capabilities.getHelloVersion());
                break;
            }
        }

        buffer.flip();  // reset the buffer position and limitation
```

4. **Get the requested server name from the explored capabilities.**

   ```
   List<SNIServerName> serverNames = capabilities.getServerNames();
   ```

5. **Look for the registered server name handler for this server name indication.**

   If the service of the host name is resident in a virtual machine or another distributed system, then the application must forward the connection to the destination. The application will need to read and write the raw internet data, rather then the SSL application from the socket stream.

   ```
   Socket destinationSocket = new Socket(serverName, 443);
   // Forward buffered bytes and network data from the current socket to the
   destinationSocket.
   ```

   If the service of the host name is resident in the same process, and the host name service can use the `SSLEngine` directly, then the application will simply feed the net data to the `SSLEngine` instance:

   ```
   // Get service context from registered handler
   // or create the context
   SSLContext serviceContext = ...

   SSLEngine serviceEngine = serviceContext.createSSLEngine();
   // Now the service can use the buffered bytes and other byte buffer as
   usual.
   ```

No SNI Extension Available

If there is no server name indication in a ClientHello message, then there is no way to select the proper service according to SNI. For such cases, the application may need to specify a default service, so that the connection can be delegated to it if there is no server name indication.

Failover SSLContext

The `SSLExplorer.explore()` method does not check the validity of TLS/DTLS contents. If the record format does not comply with TLS/DTLS specification, or the `explore()` method is

Copyright © 1993, 2026, Oracle and/or its affiliates.

invoked after handshaking has started, then the method may throw an `IOException` and be unable to produce network data. In such cases, handle the exception thrown by `SSLExplorer.explore()` by using a failover `SSLContext`, which is not used to negotiate a TLS/DTLS connection, but to close the connection with the proper alert message. The following example illustrates a failover `SSLContext`. You can find an example of the `DenialSNIMatcher` class in Case 2 in Typical Server-Side Usage Examples.

```
byte[] buffer = ...       // buffered network data
boolean failed = true;    // SSLExplorer.explore() throws an exception

SSLContext context = SSLContext.getInstance("TLS");
// the failover SSLContext

context.init(null, null, null);
SSLSocketFactory sslsf = context.getSocketFactory();
ByteArrayInputStream bais = new ByteArrayInputStream(buffer, 0, position);
SSLSocket sslSocket = (SSLSocket)sslsf.createSocket(socket, bais, true);

SNIMatcher matcher = new DenialSNIMatcher();
Collection<SNIMatcher> matchers = new ArrayList<>(1);
matchers.add(matcher);
SSLParameters params = sslSocket.getSSLParameters();
params.setSNIMatchers(matchers);    // no recognizable server name
sslSocket.setSSLParameters(params);

try {
    InputStream sslIS = sslSocket.getInputStream();
    sslIS.read();
} catch (Exception e) {
    System.out.println("Server exception " + e);
} finally {
    sslSocket.close();
}
```

Provider Pluggability

JSSE is fully pluggable and does not restrict the use of third-party JSSE providers in any way.

Transport Layer Security (TLS) Protocol Overview

Transport Layer Security (TLS) is the most widely used protocol for implementing cryptography on the web. TLS uses a combination of cryptographic processes to provide secure communication over a network. This section provides an introduction to TLS and the cryptographic processes it uses.

TLS provides a secure enhancement to the standard TCP/IP sockets protocol used for Internet communications. As shown in Table 8-10, the secure sockets layer is added between the transport layer and the application layer in the standard TCP/IP protocol stack. The application most commonly used with TLS is Hypertext Transfer Protocol (HTTP), the protocol for Internet web pages. Other applications, such as Net News Transfer Protocol (NNTP), Telnet, Lightweight Directory Access Protocol (LDAP), Interactive Message Access Protocol (IMAP), and File Transfer Protocol (FTP), can be used with TLS as well.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 8-10 TCP/IP Protocol Stack with TLS

TCP/IP Layer	Protocol
Application Layer	HTTP, NNTP, Telnet, FTP, and so on
Transport Layer Security	TLS
Transmission Control Protocol	TCP
Internet Layer	IP

Secure Socket Layer (SSL) was developed by Netscape in 1994, and with input from the Internet community, has evolved to become a standard. It is now under the control of the international standards organization, the Internet Engineering Task Force (IETF). The IETF renamed SSL to TLS, and released the first specification, version 1.0, in January 1999. TLS 1.0 is a modest upgrade to the most recent version of SSL, version 3.0. This upgrade corrected defects in previous versions and prohibited the use of known weak algorithms. TLS 1.1 was released in April 2006, TLS 1.2 in August 2008, and TLS 1.3 in August 2018. TLS 1.3 is a major overhaul of the TLS protocol and provides significant security and performance improvements over previous versions.

How TLS Works

One of the reasons that TLS is effective is that it uses several different cryptographic processes. TLS uses public-key cryptography to provide authentication, and secret-key cryptography with hash functions to provide for privacy and data integrity. Before you can understand TLS, it's helpful to understand these cryptographic processes.

Cryptographic Processes

The primary purpose of cryptography is to make it difficult for an unauthorized third party to access and understand private communication between two parties. It is not always possible to restrict all unauthorized access to data, but private data can be made unintelligible to unauthorized parties through the process of encryption. Encryption uses complex algorithms to convert the original message (cleartext) to an encoded message (ciphertext). The algorithms used to encrypt and decrypt data that is transferred over a network typically come in two categories: secret-key cryptography and public-key cryptography.

Both secret-key cryptography and public-key cryptography depend on the use of an agreed-upon cryptographic key or pair of keys. A key is a string of bits that is used by the cryptographic algorithm or algorithms during the process of encrypting and decrypting the data. A cryptographic key is like a key for a lock; only with the right key can you open the lock.

Safely transmitting a key between two communicating parties is not a trivial matter. A public key certificate enables a party to safely transmit its public key, while providing assurance to the receiver of the authenticity of the public key. See Public Key Certificates.

The descriptions of the cryptographic processes in secret-key cryptography and public-key cryptography follow conventions widely used by the security community: the two communicating parties are labeled with the names Alice and Bob. The unauthorized third party, also known as the attacker, is named Charlie.

Secret-Key Cryptography

With secret-key cryptography, both communicating parties, Alice and Bob, use the same key to encrypt and decrypt the messages. Before any encrypted data can be sent over the network,

Copyright © 1993, 2026, Oracle and/or its affiliates.

both Alice and Bob must have the key and must agree on the cryptographic algorithm that they will use for encryption and decryption

One of the major problems with secret-key cryptography is the logistical issue of how to get the key from one party to the other without allowing access to an attacker. If Alice and Bob are securing their data with secret-key cryptography, and if Charlie gains access to their key, then Charlie can understand any secret messages he intercepts between Alice and Bob. Not only can Charlie decrypt Alice's and Bob's messages, but he can also pretend that he is Alice and send encrypted data to Bob. Bob won't know that the message came from Charlie, not Alice.

After the problem of secret key distribution is solved, secret-key cryptography can be a valuable tool. The algorithms provide excellent security and encrypt data relatively quickly. The majority of the sensitive data sent in an TLS session is sent using secret-key cryptography.

Secret-key cryptography is also called symmetric cryptography because the same key is used to both encrypt and decrypt the data. Well-known secret-key cryptographic algorithms include Advanced Encryption Standard (AES), Triple Data Encryption Standard (3DES), and Rivest Cipher 4 (RC4).

Public-Key Cryptography

Public-key cryptography solves the logistical problem of key distribution by using both a public key and a private key. The public key can be sent openly through the network while the private key is kept private by one of the communicating parties. The public and the private keys are cryptographic inverses of each other; what one key encrypts, the other key will decrypt.

Assume that Bob wants to send a secret message to Alice using public-key cryptography. Alice has both a public key and a private key, so she keeps her private key in a safe place and sends her public key to Bob. Bob encrypts the secret message to Alice using Alice's public key. Alice can later decrypt the message with her private key.

If Alice encrypts a message using her private key and sends the encrypted message to Bob, then Bob can be sure that the data he receives comes from Alice; if Bob can decrypt the data with Alice's public key, the message must have been encrypted by Alice with her private key, and only Alice has Alice's private key. The problem is that anybody else can read the message as well because Alice's public key is public. Although this scenario does not allow for secure data communication, it does provide the basis for digital signatures. A digital signature is one of the components of a public key certificate, and is used in TLS to authenticate a client or a server. See Public Key Certificates and Digital Signatures.

Public-key cryptography is also called *asymmetric cryptography* because different keys are used to encrypt and decrypt the data. A well-known public key cryptographic algorithm often used with TLS is the Rivest Shamir Adleman (RSA) algorithm. Another public key algorithm used with TLS that is designed specifically for secret key exchange is the Diffie-Hellman (DH) algorithm. Public-key cryptography requires extensive computations, making it very slow. It is therefore typically used only for encrypting small pieces of data, such as secret keys, rather than for the bulk of encrypted data communications.

Comparison Between Secret-Key and Public-Key Cryptography

Both secret-key cryptography and public-key cryptography have strengths and weaknesses. With secret-key cryptography, data can be encrypted and decrypted quickly, but because both communicating parties must share the same secret key information, the logistics of exchanging the key can be a problem. With public-key cryptography, key exchange is not a problem because the public key does not need to be kept secret, but the algorithms used to encrypt and decrypt data require extensive computations, and are therefore very slow.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Public Key Certificates

A public key certificate provides a safe way for an entity to pass on its public key to be used in asymmetric cryptography. The public key certificate avoids the following situation: if Charlie creates his own public key and private key, he can claim that he is Alice and send his public key to Bob. Bob will be able to communicate with Charlie, but Bob will think that he is sending his data to Alice.

A public key certificate can be thought of as the digital equivalent of a passport. It is issued by a trusted organization and provides identification for the bearer. A trusted organization that issues public key certificates is known as a Certificate Authority (CA). The CA can be likened to a notary public. To obtain a certificate from a CA, one must provide proof of identity. Once the CA is confident that the applicant represents the organization it says it represents, the CA signs the certificate attesting to the validity of the information contained within the certificate.

A public key certificate contains the following fields:

Issuer
The CA that issued the certificate. If a user trusts the CA that issued the certificate, and if the certificate is valid, then the user can trust the certificate.

Period of validity
A certificate has an expiration date. This date should be checked when verifying the validity of a certificate.

Subject
Includes information about the entity that the certificate represents.

Subject's public key
The primary piece of information that the certificate provides is the subject's public key. All the other fields are provided to ensure the validity of this key.

Signature
The certificate is digitally signed by the CA that issued the certificate. The signature is created using the CA's private key and ensures the validity of the certificate. Because only the certificate is signed, not the data sent in the TLS transaction, TLS does not provide for nonrepudiation.

If Bob only accepts Alice's public key as valid when she sends it in a public key certificate, then Bob won’t be fooled into sending secret information to Charlie when Charlie masquerades as Alice.

Multiple certificates may be linked in a certificate chain. When a certificate chain is used, the first certificate is always that of the sender. The next is the certificate of the entity that issued the sender's certificate. If more certificates are in the chain, then each is that of the authority that issued the previous certificate. The final certificate in the chain is the certificate for a root CA. A root CA is a public Certificate Authority that is widely trusted. Information for several root CAs is typically stored in the client's Internet browser. This information includes the CA's public key. Well-known CAs include Comodo, DigiCert, and GoDaddy.

Cryptographic Hash Functions

When sending encrypted data, TLS typically uses a cryptographic hash function to ensure data integrity. The hash function prevents Charlie from tampering with data that Alice sends to Bob.

Copyright © 1993, 2026, Oracle and/or its affiliates.

A cryptographic hash function is similar to a checksum. The main difference is that whereas a checksum is designed to detect accidental alterations in data, a cryptographic hash function is designed to detect deliberate alterations. When data is processed by a cryptographic hash function, a small string of bits, known as a hash, is generated. The slightest change to the message typically makes a large change in the resulting hash. A cryptographic hash function does not require a cryptographic key. A hash function often used with TLS is Secure Hash Algorithm (SHA). SHA was proposed by the U.S. National Institute of Standards and Technology (NIST).

Message Authentication Code

A message authentication code (MAC) is similar to a cryptographic hash, except that it is based on a secret key. When secret key information is included with the data that is processed by a cryptographic hash function, then the resulting hash is known as an HMAC.

If Alice wants to be sure that Charlie does not tamper with her message to Bob, then she can calculate an HMAC for her message and append the HMAC to her original message. She can then encrypt the message plus the HMAC using a secret key that she shares with Bob. When Bob decrypts the message and calculates the HMAC, he will be able to tell if the message was modified in transit. With TLS, an HMAC is used with the transmission of secure data.

Digital Signatures

Once a cryptographic hash is created for a message, the hash is encrypted with the sender's private key. This encrypted hash is called a digital signature.

The TLS 1.3 Handshake

Communication using TLS 1.3 begins the TLS handshake. This is an initial negotiation between the client and server that establishes the parameters of their subsequent interactions within TLS. It consists of three phases: key exchange, server parameters, and authentication:

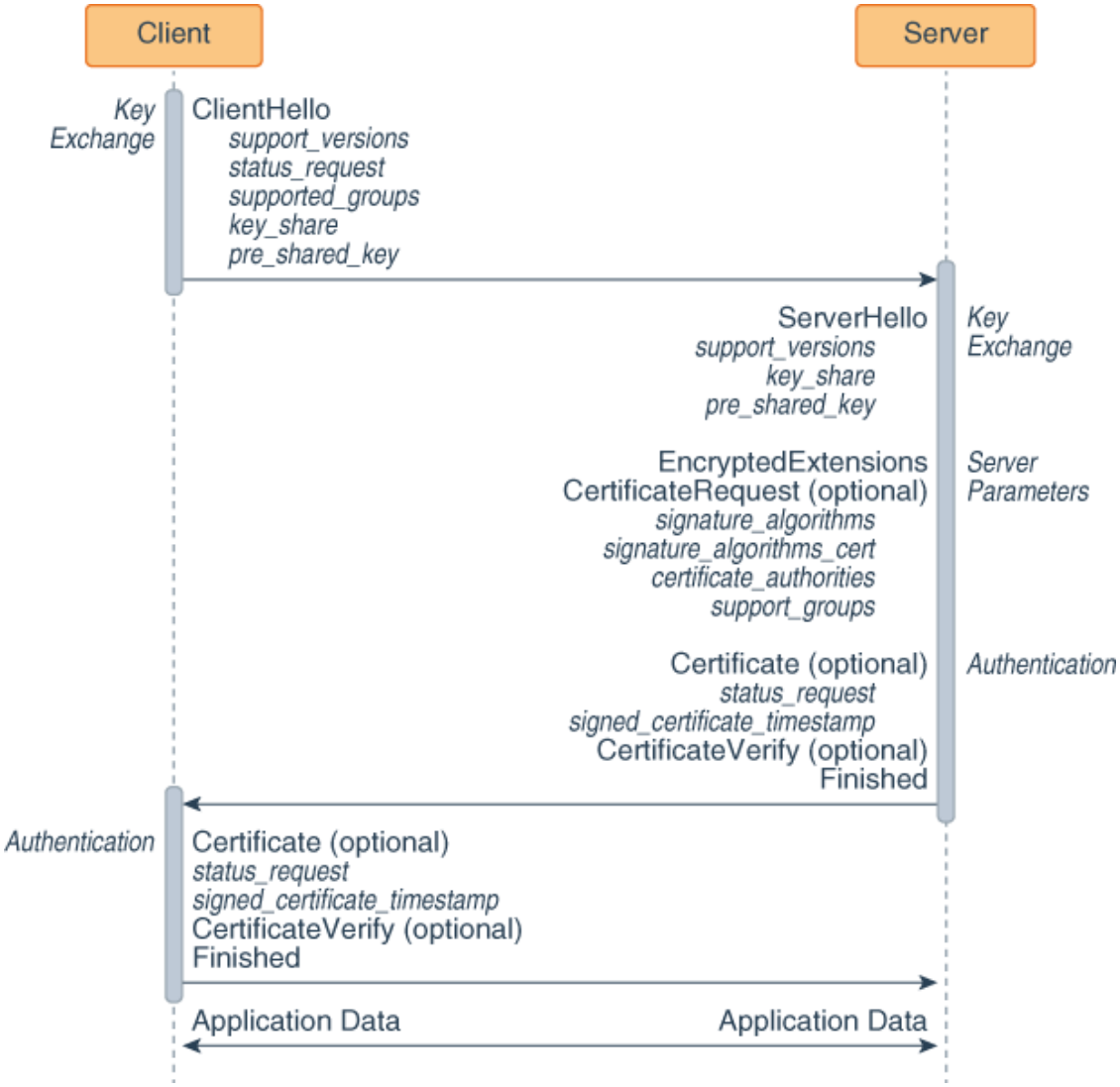
1. **Key Exchange**: This phase establishes shared keying material, such as which named group the shared key can belong to (Elliptic Curve Groups (ECDHE) or Finite Field Groups (DHE)), and selects cryptographic parameters, such as symmetric cipher options.
2. **Server Parameters**: This phase establishes other handshake parameters such as whether certificate-based client authentication is desired.
3. **Authentication**: This phase authenticates the server (and optionally the client) and provides key confirmation and handshake integrity.

The TLS 1.3 Protocol

The following figure shows the sequence of messages for the full TLS handshake.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 8-7 TLS 1.3 Handshake



1. Key exchange:
   a. The client sends a ClientHello message to server.
   b. The server processes the ClientHello message and determines the appropriate cryptographic parameters for the connection. It then responds with its own ServerHello message, which indicates the negotiated connection parameters. For TLS 1.3, the ServerHello message determines the key and cipher options only. Other handshake parameters may be determined later.
2. Server parameters: The server sends two messages to establish server parameters:
   - EncryptedExtensions: This message contains responses to ClientHello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates.
   - CertificateRequest (optional): If certificate-based client authentication is desired, then the server sends this message, which contains the desired parameters for that certificate. This message is omitted if client authentication is not desired.
3. Authentication:

Copyright © 1993, 2026, Oracle and/or its affiliates.

a. The server sends these authentication messages:
   - Certificate (optional): This message contains the authentication certificate and any other supporting certificates in the certificate chain. This message is omitted if the server is not authenticating with a certificate.

     > **Note**
     >
     > The Certificate message can contain a raw key instead of a certificate.

   - CertificateVerify (optional): This message contains a signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message is omitted if the server is not authenticating with a certificate.
   - Finished: a MAC (Message Authentication Code) over the entire handshake.

b. The client responds with its own Certificate, CertificateVerify, and Finished messages. The Certificate message is omitted if the server did not send a CertificateRequest message. The CertificateVerify message is omitted if the client is not authenticating with a certificate.

The client and server can now securely send application data to each other.

Key Exchange

The key exchange messages, ClientHello and ServerHello, determine the security capabilities of the client and the server and establish shared secrets, including the traffic keys used to protect the rest of the handshake and the application data.

ClientHello

The TLS handshake begins with the client sending a ClientHello message to the server. This message contains the following fields:

> **Note**
>
> TLS messages may contain additional fields than the ones listed here; see the TLS 1.3 specification for full details about TLS messages and their fields.

- **cipher_suites**: This field contains a list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with Keyed-Hash Message Code (HMAC)-based Extract-and-Expand Key Derivation Function (HKDF).
- **extensions**: Extensions facilitates the addition of new features to the TLS protocol with minimal impact to existing clients. Extensions that the ClientHello message may contain, but are not limited to, the following:
  - **supported_versions**: This extension indicates which versions of TLS the client supports. The ClientHello message must contain this message.
  - **status_request**: This extension indicates that client wants to use a certificate status protocol; the server may not agree to use it. An example of a certificate status protocol is Online Certificate Status Protocol (OCSP). See Client-Driven OCSP and OCSP Stapling.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- **supported_groups**: This extension indicates the named groups that the client supports for key exchange. These named groups include elliptic curve groups (ECDHE) and finite field groups (DHE). The ClientHello message must include this message if it's using ECDHE or DHE key exchange.
- **key_share**: This extension contains a list of cryptographic parameters for key exchange. It contains a field named client_shares that contains this list. Each item in this list contains the following fields:
  * **group**: The name of the group on which the key exchange cryptographic method is based. See The SunJSSE Provider in JDK Providers Documentation.
  * **key_exchange**: Key exchange information, which is determined by the value of the group field.
- **pre_shared_key**: A pre-shared key (PSK) is a shared secret that was previously shared between the two parties using some secure channel before it needs to be used. PSKs can be established in a previous connection and then used to establish a new connection. Once a handshake has completed, the server can send to the client a PSK identity that corresponds to a unique key derived from the initial handshake. See Session Resumption with a Pre-Shared Key.
- **cookie**: When a server sends a HelloRetryRequest message, it can include this extension to the client. (The server sends a HelloRetryRequest message in response to a ClientHello message if it can find an acceptable set of parameters, but the ClientHello message doesn't have enough information to proceed with the handshake.) One purpose of this extension is to enable the server to force the client to demonstrate reachability at their apparent network address (which provides some denial-of-service attack (DoS) protection. When the client sends a new ClientHello message, it must copy the contents received in the HelloRetryRequest into a cookie extension in this new ClientHello message.
- **server_name**: TLS 1.3 doesn't provide a mechanism for a client to tell a server the name of the server it is contacting. Clients can use this extension to provide this information to facilitate connections to servers that host multiple virtual servers at a single network address. Note that some servers may require clients to send this extension.

ServerHello

The server responds to the client's ClientHello message with a ServerHello message if it's able to negotiate an acceptable set of handshake parameters. This message contains the following fields:

- **cipher_suite**: This field contains the single cipher suite selected by the server from the list in the ClientHello.cipher_suites field.
- **extensions**: This field contains extensions that are required to establish the cryptographic context and negotiate the protocol version. The extensions that the SeverHello may contain include the following:
  - **supported_versions**: Indicates which version of TLS it's using. The ServerHello message must contain this extension.
  - **key_share**: This extension contains a list of cryptographic parameters for key exchange.
  - **pre_shared_key**: This extension contains the pre-shared key the server agreed to use. See Session Resumption with a Pre-Shared Key for information about pre-shared keys.

  The server sends other extensions separately in the EncryptedExtensions message.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Server Parameters

After the server sends a ServerHello message to the client, it sends two messages to establish server parameters: EncryptedExtensions and CertificateRequest:

- **EncryptedExtensions**: This message contains responses to ClientHello extensions that are not required to determine cryptographic parameters other than those that are specific to individual certificates.
- **CertificateRequest**: If certificate-based client authentication is desired, then this message is sent. It contains parameters for a certificate requested from the client. It includes the following fields:
  - **certificate_request_context**: This field contains an identifier that identifies the certificate request
  - **extensions**: This field contains extensions that describe the requested certificate's parameters. It may contain the following extensions:
  - **signature_algorithms**: This extension indicates which signature algorithms may be used in CertificateVerify messages. The ServerHello message must contain this extension.
  - **signature_algorithms_cert**: This extension indicates which signature algorithms may be used in digital signatures. If this message isn't sent, then it uses the values specified in the signature_algorithms extension.
  - **certificate_authorities**: This extension indicates which certificate authorities the server accepts.
  - **supported_groups**: This message contains named groups that the server prefers. The client may use this information to change what groups it uses in its key_share extension in subsequent connections.

Authentication

The last three messages that the server and client send to each other in a TLS handshake are Certificate, CertificateVerify, and Finished.

Certificate

This message contains the authentication certificate and any other supporting certificates in the certificate chain. The server must send this message if the key exchange method uses certificates for authentication. The client must send this if and only if the server requested client authentication through a CertificateRequest message. The certificate message includes the following fields:

- **certificate_list**: This field contains a sequence of CertificateEntry structures, each containing a single certificate and a set of extensions
- **extensions**: Extensions that the Certificate message may contain include the following:
  - **status_request**: See Client-Driven OCSP and OCSP Stapling
  - **signed_certificate_timestamp**: TLS clients won't accept certificates unless they are logged. When a valid certificate is submitted to a log, the log must return a Signed Certificate Timestamp (SCT); see RFC 6962: Certificate Transparency.

Copyright © 1993, 2026, Oracle and/or its affiliates.

CertificateVerify

This message contains a signature over the entire handshake using the private key corresponding to the public key in the Certificate message. It provides proof that the client or the server has the private key corresponding to its certificate. This message includes the following fieds:

- **algorithm**: This field contains the signature algorithm used. See The SunJSSE Provider in JDK Providers Documentation for supported algorithms.
- **signature**: This field contains the digital signature using the algorithm.

Finished

This message contains a Message Authentication Code (MAC) over the entire handshake. Once the client and server have verified the Finished messages that they have received from their peers, both sides may send and receive application data over the connection.

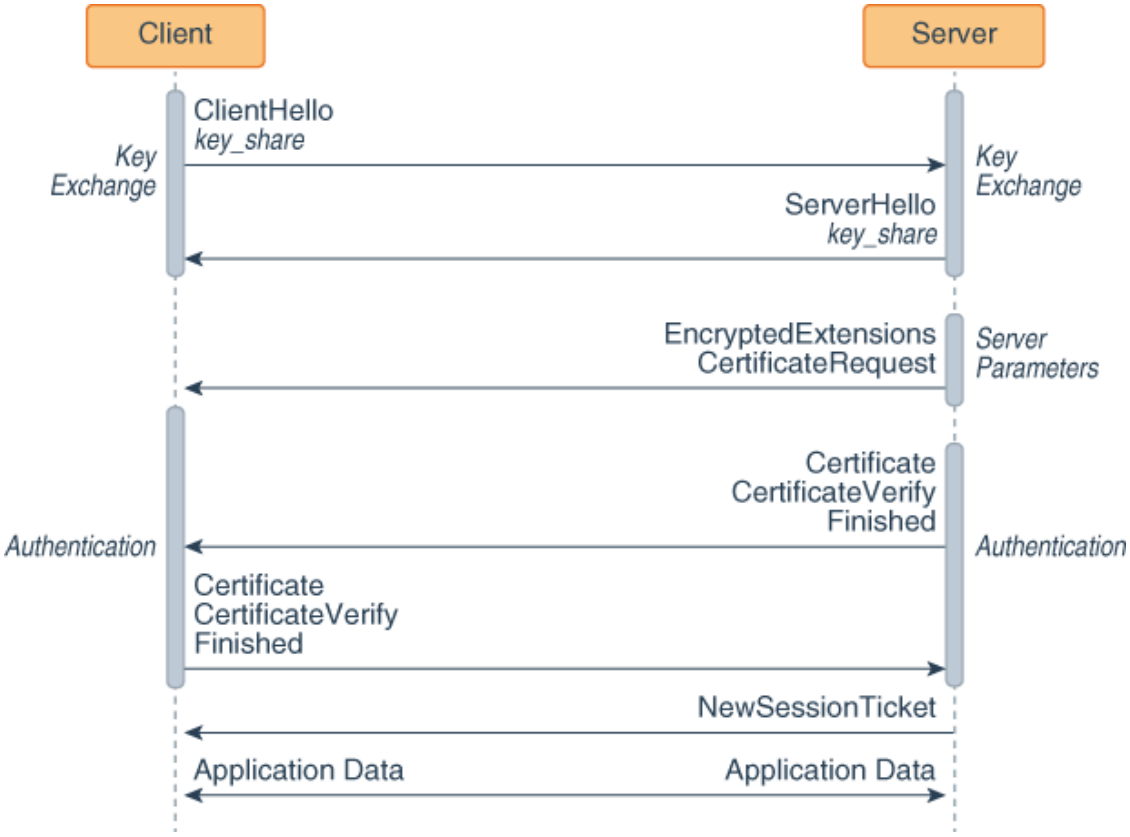
Session Resumption with a Pre-Shared Key

A pre-shared key (PSK) is a shared secret that was previously shared between the two parties using some secure channel before it needs to be used. You can establish a PSK during one TLS handshake and then use it to establish a new connection in another handshake; this is called session resumption with a PSK. The PSK corresponds to a unique key derived from the initial handshake. If the server accepts the PSK when establishing a new connection, then the security context of this connection is cryptographically tied to the original connection, and the key derived from the initial handshake is used to bootstrap the cryptographic state instead of the full TLS handshake.

The following figures show two handshakes, the first establishes a PSK and the second uses it.

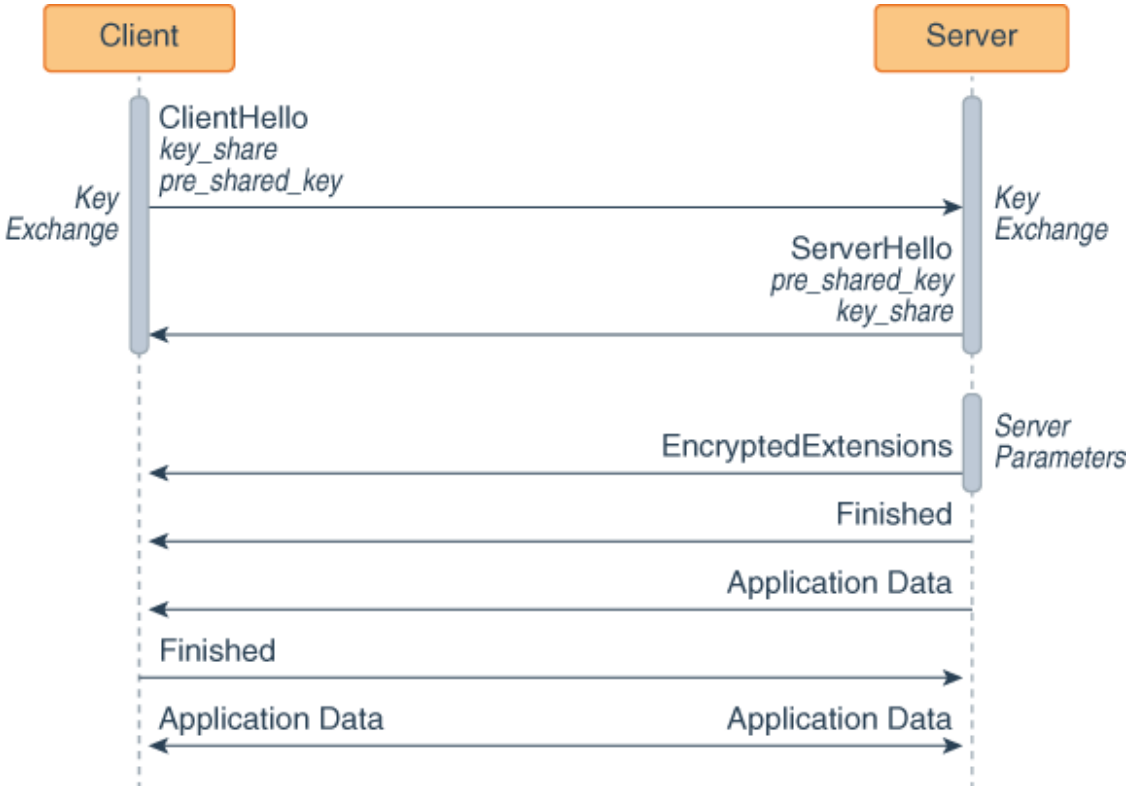
Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 8-8 TLS 1.3 Handshake That Establishes a PSK



Copyright © 1993, 2026, Oracle and/or its affiliates.

Figure 8-9 TLS 1.3 Handshake That Uses a PSK



1. The client sends a ClientHello message with a key_share extension to the server. This extension lists which key exchange cryptographic methods that the client supports.
2. The server responds with a ServerHello message with a key_share extension. This extension contains the cryptographic method it wants to use for the key exchange.
3. The server sends its server parameters to the client.
4. Both the server and client exchange authentication messages.
5. The server sends a NewSessionTicket message to the client, which contains a PSK that the client then may use for future handshakes by including it in the pre_shared_key extension in its ClientHello message.
6. The client and server can now exchange encrypted application data.
7. In a future handshake, the client sends to the server a ClientHello message with the key_share and pre_shared_key extensions. The pre_shared_key extension contains a PSK sent in a NewSessionTicket message.
8. The server responds with a ServerHello message with the pre_shared_key and key_share extensions. The pre_shared_key extension contains the PSK the server as agreed to use.
9. The server sends its parameters to the client.
10. The server and the client send each other Finished messages. They don't perform the authentication phase as the security context of this connection is cryptographically tied to the original connection.
11. The client and server can now exchange encrypted application data.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The following are not supported in JDK 11:
>
> - Resumption using PSK only: You must use PSKs with (EC)DHE key exchange, which provides forward secrecy in combination with shared keys. Resumption using PSK only is less secure with regards to forward and backward secrecy.
> - Zero Round Trip Time Resumption (0–RTT): This enables the client and server to send application data in the first messages (ClientHello and ServerHello) to each other. The client uses a PSK to encrypt the application data it initially sends with the ClientHello and to authenticate the server. This has the security issues of resumption using PSK only and some potential for replay attacks.
> - Stateless server PSKs: RFC5077: Transport Layer Security (TLS) Session Resumption without Server-Site State describes a mechanism that enables the server to resume sessions and avoid keeping per-client session state. This mechanism would reduce server memory usage at the expense of forward secrecy for resumption using PSK only.
> - Out-of-band PSK establishment: This means the production of PSKs other than through NewSessionTicket messages.

Post-Handshake Messages

The client and server can send other messages after the handshake: new session ticket message, post-handshake authentication, and key update.

New Session Ticket Message

The NewSessionTicket message, sent by the server after it receives the Finished message, contains a pre-shared key that the client then may use for future handshakes. See Session Resumption with a Pre-Shared Key.

Post-Handshake Authentication

If client sent the post_handshake_auth extension, the server may request client authentication at any time after the handshake by sending a CertificateRequest message. If the client authenticates, then it must send Certificate, CertificateVerify, and Finished messages. If the client declines, then it must send a Certificate message that contains no certificates and the Finished message.

KeyUpdate Message

The KeyUpdate handshake message is used to indicate that the sender is updating its sending cryptographic keys. It replaces the ChangeCipherSpec message in TLS 1.2.

You can specify a limit on the amount of data an algorithm may encrypt with a specific set of keys with the `jdk.tls.keyLimits` Security Property. See Limiting Amount of Data Algorithms May Encrypt with a Set of Keys.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Compatibility Risks and Known Issues

Enhancements to JSSE may introduce compatibility problems and other known issues, which are described in this section.

TLS 1.3 Not Directly Compatible with Previous Versions

TLS 1.3 is not directly compatible with previous versions. Although TLS 1.3 can be implemented with a backward-compatibility mode, there are still several compatibility risks to consider when upgrading to TLS 1.3:

- TLS 1.3 uses a half-close policy, while TLS 1.2 and earlier use a duplex-close policy. For applications that depend on the duplex-close policy, there may be compatibility issues when upgrading to TLS 1.3.
- The signature_algorithms_cert extension requires that pre-defined signature algorithms are used for certificate authentication. In practice, however, an application may use unsupported signature algorithms.
- The DSA signature algorithm is not supported in TLS 1.3. If a server is configured to only use DSA certificates, it cannot negotiate a TLS 1.3 connection.
- The supported cipher suites for TLS 1.3 are not the same as TLS 1.2 and earlier. If an application hardcodes cipher suites that are no longer supported, it may not be able to use TLS 1.3 without modifications to its code, for example TLS_AES_128_GCM_SHA256 (1.3 and later) versus TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (1.2 and earlier).
- The TLS 1.3 session resumption and key update behaviors are different from TLS 1.2 and earlier. The compatibility impact should be minimal, but it could be a risk if an application depends on the handshake details of the TLS protocols.

The TLS 1.2 Handshake

Communication using SSL begins with an exchange of information between the client and the server. This exchange of information is called the SSL handshake. The SSL handshake includes the following stages:

1. **Negotiating the cipher suite**
   The SSL session begins with a negotiation between the client and the server as to which cipher suite they will use. A cipher suite is a set of cryptographic algorithms and key sizes that a computer can use to encrypt data. The cipher suite includes information about the public key exchange algorithms or key agreement algorithms, and cryptographic hash functions. The client tells the server which cipher suites it has available, and the server chooses the best mutually acceptable cipher suite.
2. **Authenticating the server's identity (optional)**
   In SSL, the authentication step is optional, but in the example of an e-commerce transaction over the web, the client will generally want to authenticate the server. Authenticating the server allows the client to be sure that the server represents the entity that the client believes the server represents.

   To prove that a server belongs to the organization that it claims to represent, the server presents its public key certificate to the client. If this certificate is valid, then the client can be sure of the identity of the server.

   The client and server exchange information that allows them to agree on the same secret key. For example, with RSA, the client uses the server's public key, obtained from the public key certificate, to encrypt the secret key information. The client sends the encrypted

Copyright © 1993, 2026, Oracle and/or its affiliates.

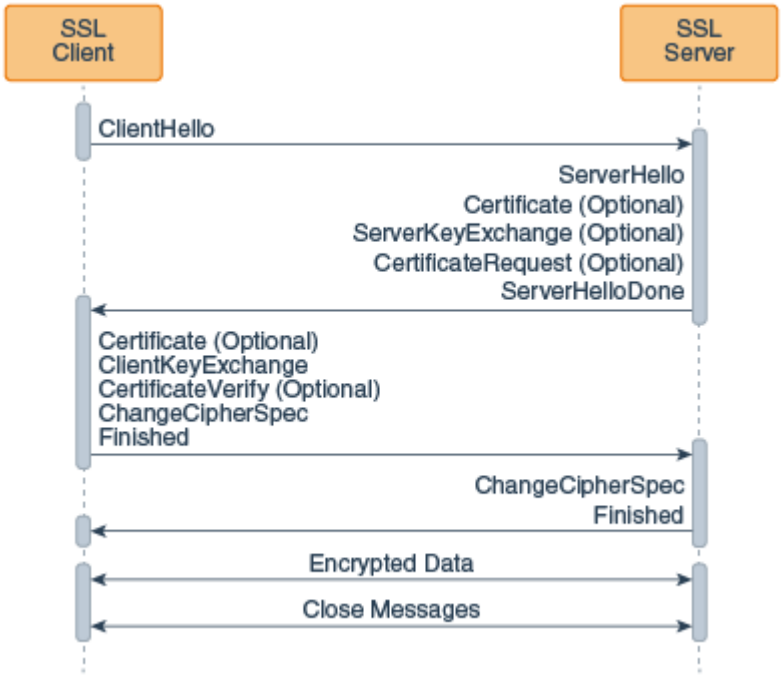
secret key information to the server. Only the server can decrypt this message because the server's private key is required for this decryption.

3. **Agreeing on encryption mechanisms**
   Both the client and the server now have access to the same secret key. With each message, they use the cryptographic hash function, chosen in the first step of the handshake, and shared secret information, to compute an HMAC that they append to the message. They then use the secret key and the secret key algorithm negotiated in the first step of the handshake to encrypt the secure data and the HMAC. The client and server can now communicate securely using their encrypted and hashed data.

The TLS 1.2 Protocol

The TLS 1.2 Handshake provides a high-level description of the SSL handshake, which is the exchange of information between the client and the server prior to sending the encrypted message. Figure 8-10 provides more detail. It shows the sequence of messages that are exchanged in the SSL handshake. Messages that are sent only in certain situations are noted as optional. Each of the SSL messages is described in detail afterward.

Figure 8-10 The SSL/TLS Handshake



The SSL messages are sent in the following order:

1. **Client hello:** The client sends the server information including the highest version of SSL that it supports and a list of the cipher suites that it supports (TLS 1.0 is indicated as SSL 3.1). The cipher suite information includes cryptographic algorithms and key sizes.
2. **Server hello:** The server chooses the highest version of SSL and the best cipher suite that both the client and server support and sends this information to the client.
3. (Optional) **Certificate:** The server sends the client a certificate or a certificate chain. A certificate chain typically begins with the server's public key certificate and ends with the

Copyright © 1993, 2026, Oracle and/or its affiliates.

certificate authority's root certificate. This message is optional, but is used whenever server authentication is required.

4. (Optional) **Certificate request:** If the server must authenticate the client, then it sends the client a certificate request. In Internet applications, this message is rarely sent.
5. (Optional) **Server key exchange:** The server sends the client a server key exchange message if the public key information from the Certificate is not sufficient for key exchange. For example, in cipher suites based on Diffie-Hellman (DH), this message contains the server's DH public key.
6. **Server hello done:** The server tells the client that it is finished with its initial negotiation messages.
7. (Optional) **Certificate:** If the server Certificate request from the client, the client sends its certificate chain, just as the server did previously.

> **Note**
>
> Only a few Internet server applications ask for a certificate from the client.

8. **Client key exchange:** The client generates information used to create a key to use for symmetric encryption. For RSA, the client then encrypts this key information with the server's public key and sends it to the server. For cipher suites based on DH, this message contains the client's DH public key.
9. (Optional) **Certificate verify:** This message is sent by the client when the client presents a certificate as previously explained. Its purpose is to allow the server to complete the process of authenticating the client. When this message is used, the client sends information that it digitally signs using a cryptographic hash function. When the server decrypts this information with the client's public key, the server is able to authenticate the client.
10. **Change cipher spec:** The client sends a message telling the server to change to encrypted mode.
11. **Finished** The client tells the server that it is ready for secure data communication to begin.
12. **Change cipher spec:** The server sends a message telling the client to change to encrypted mode.
13. **Finished:** The server tells the client that it is ready for secure data communication to begin. This is the end of the SSL handshake.
14. **Encrypted data:** The client and the server communicate using the symmetric encryption algorithm and the cryptographic hash function negotiated during the client hello and server hello, and using the secret key that the client sent to the server during the client key exchange. The handshake can be renegotiated at this time. See Handshaking Again (Renegotiation).
15. **Close Messages:**At the end of the connection, each side sends a `close_notify` alert to inform the peer that the connection is closed.

If the parameters generated during an SSL session are saved, then these parameters can sometimes be reused for future SSL sessions. Saving SSL session parameters allows encrypted communication to begin much more quickly.

Handshaking Again (Renegotiation)

Once the initial handshake is finished and application data is flowing, either side is free to initiate a new handshake at any time. An application might like to use a stronger cipher suite

Copyright © 1993, 2026, Oracle and/or its affiliates.

for especially critical operations, or a server application might want to require client authentication.

Regardless of the reason, the new handshake takes place over the existing encrypted session, and application data and handshake messages are interleaved until a new session is established.

Your application can initiate a new handshake by using one of the following methods:

- `SSLSocket.startHandshake()`
- `SSLEngine.beginHandshake()`

Cipher Suite Choice and Remote Entity Verification

The SSL/TLS protocols define a specific series of steps to ensure a *protected* connection. However, the choice of cipher suite directly affects the type of security that the connection enjoys. For example, if an anonymous cipher suite is selected, then the application has no way to verify the remote peer's identity. If a suite with no encryption is selected, then the privacy of the data cannot be protected. Additionally, the SSL/TLS protocols do not specify that the credentials received must match those that peer might be expected to send. If the connection were somehow redirected to a rogue peer, but the rogue's credentials were acceptable based on the current trust material, then the connection would be considered valid.

When using raw `SSLSocket` and `SSLEngine` classes, you should always check the peer's credentials before sending any data. The `SSLSocket` and `SSLEngine` classes do not automatically verify that the host name matches the host name in the peer's credentials. An application could be exploited with host name spoofing if the host name is not verified. Since JDK 7, endpoint identification/verification procedures can be handled during SSL/TLS handshaking. See the `SSLParameters.getEndpointIdentificationAlgorithm` method.

Protocols such as HTTPS (HTTP Over TLS) do require host name verification. Since JDK 7, the HTTPS endpoint identification is enforced during handshaking for `HttpsURLConnection` by default. See the `SSLParameters.getEndpointIdentificationAlgorithm` method. Alternatively, applications can use the `HostnameVerifier` interface to override the default HTTPS host name rules. See HostnameVerifier Interface and HttpsURLConnection Class.

Datagram Transport Layer Security (DTLS) Protocol

Datagram Transport Layer Security (DTLS) protocol is designed to construct “TLS over datagram” traffic that doesn't require or provide reliable or in-order delivery of data. Java Secure Socket Extension (JSSE) API and the SunJSSE security provider support the DTLS protocol.

Because the TLS requires a transparent reliable transport channel such as TCP it can’t be used to secure unreliable datagram traffic. DTLS is a datagram-compatible variant of TLS.

The JSSE API now supports DTLS Version 1.0 and DTLS Version 1.2 along with Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols.

The `javax.net.ssl.SSLEngine` programming model is used by the JSSE API for DTLS.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The DTLS Handshake

Before application data can be sent or received, the DTLS protocol requires a handshake to establish cryptographic parameters. This handshake requires a series of back-and-forth messages between the client and server by the `SSLEngine` object.

DTLS handshake requires all messages be received properly. Thus, in unreliable datagram traffic, missing or delayed packets must be retransmitted. Since `javax.net.ssl.SSLEngine` is not responsible for I/O operations, it is up to the application to provide timers and signal the SSLEngine when a retransmission is needed. It is important that you implement a timer and retransmission strategy for your application. See Handling Retransmissions in DTLS Connections.

The DTLS handshake includes the following stages:

1. **Negotiating the cipher suite**
   The DTLS session begins with a negotiation between the client and the server as to which cipher suite they will use. A cipher suite is a set of cryptographic algorithms and key sizes that a computer can use to encrypt data. The cipher suite includes information about the public key exchange algorithms or key agreement algorithms, and cryptographic hash functions. The client tells the server which cipher suites it has available, and the server chooses the best mutually acceptable cipher suite.

   A cookie is exchanged between the client and server along with the cipher suite in order to prevent denial of service attacks (DoS).

2. **Authenticating the server's identity (optional)**
   The authentication step is optional, but in the example of an e-commerce transaction over the web, the client chooses to authenticate the server. Authenticating the server allows the client to be sure that the server represents the entity that the client believes the server represents.

   To prove that a server belongs to the organization that it claims to represent, the server presents its public key certificate to the client. If this certificate is valid, then the client can be sure of the identity of the server.

   The client and server exchange information that allows them to agree on the same secret key. For example, with RSA, the client uses the server's public key, obtained from the public key certificate, to encrypt the secret key information. The client sends the encrypted secret key information to the server. Only the server can decrypt this message because the server's private key is required for this decryption.

3. **Agreeing on encryption mechanisms**
   Both the client and the server now have access to the same secret key. With each message, they use the cryptographic hash function, chosen in the first step of the handshake, and shared secret information, to compute an HMAC that they append to the message. They then use the secret key and the secret key algorithm negotiated in the first step of the handshake to encrypt the secure data and the HMAC. The client and server can now communicate securely using their encrypted and hashed data.

The DTLS Handshake Message Exchange

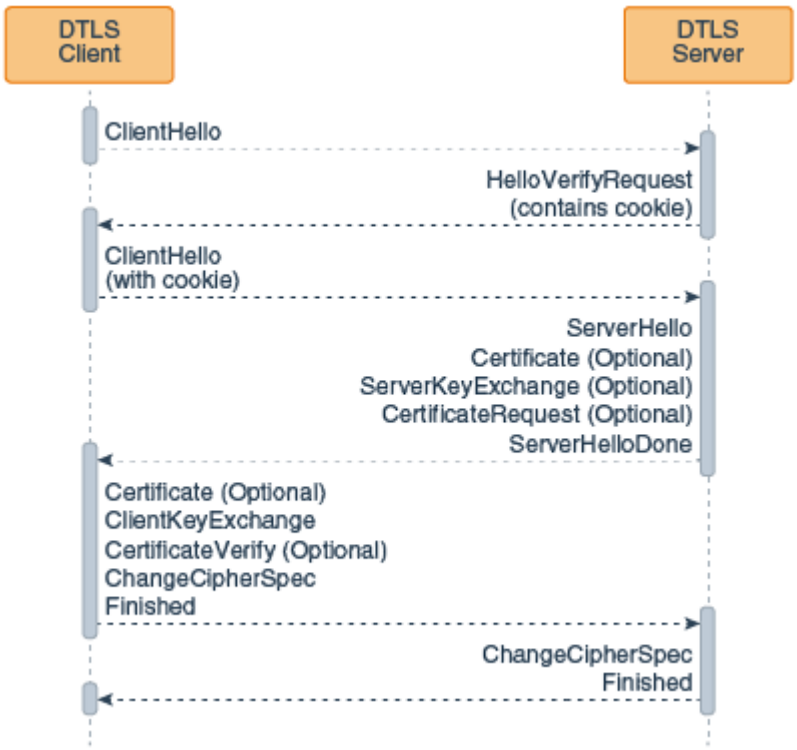
In a DTLS handshake, series of back-and-forth messages are exchanged between the client and server by the `SSLEngine` object.

Figure 8-11 shows the sequence of messages that are exchanged in the DTLS handshake. Messages that are sent only in certain situations are noted as optional. Each message is described following the figure.

Copyright © 1993, 2026, Oracle and/or its affiliates.

To know more about DTLS handshake messages, see DTLS Version 1.0 and DTLS Version 1.2.

Figure 8-11 DTLS Handshake



The following handshake messages are exchanged between the client and server during DTLS handshake:

1. **ClientHello:**
   The client sends the server information including the highest version of DTLS that it supports and a list of the cipher suites that it supports. The cipher suite information includes cryptographic algorithms and key sizes.
2. **HelloVerifyRequest:**
   The server responds to the ClientHello message from the client with a cookie.
3. **ClientHello:**
   The client sends a second ClientHello message to the server with highest version of DTLS that it supports and a list of the cipher suites that it supports. The cookie received in the HelloVerifyRequest is sent back to the server.
4. **ServerHello:**
   The server chooses the highest version of DTLS and the best cipher suite that both the client and server support and sends this information to the client.
5. (Optional) **Certificate:**
   The server sends the client a certificate or a certificate chain. A certificate chain typically begins with the server's public key certificate and ends with the certificate authority's root certificate. This message is optional, but is used whenever server authentication is required
6. (Optional) **CertificateRequest:**

Copyright © 1993, 2026, Oracle and/or its affiliates.

If the server must authenticate the client, then it sends the client a certificate request. In Internet applications, this message is rarely sent.

7. (Optional) **ServerKeyExchange:**
   The server sends the client a server key exchange message if the public key information from the Certificate is not sufficient for key exchange. For example, in cipher suites based on Diffie-Hellman (DH), this message contains the server's DH public key.
8. **ServerHelloDone:**
   The server tells the client that it is finished with its initial negotiation messages.
9. (Optional) **Certificate:**
   If the server Certificate request from the client, the client sends its certificate chain, just as the server did previously.

> **Note**
>
> Only a few Internet server applications ask for a certificate from the client.

10. **ClientKeyExchange:**
    The client generates information used to create a key to use for symmetric encryption. For RSA, the client then encrypts this key information with the server's public key and sends it to the server. For cipher suites based on DH, this message contains the client's DH public key.
11. (Optional) **CertificateVerify:**
    This message is sent by the client when the client presents a certificate as previously explained. Its purpose is to allow the server to complete the process of authenticating the client. When this message is used, the client sends information that it digitally signs using a cryptographic hash function. When the server decrypts this information with the client's public key, the server is able to authenticate the client.
12. **ChangeCipherSpec:**
    The client sends a message telling the server that subsequent data will be protected under the newly negotiated CipherSpec and keys and the data is encrypted
13. **Finished:**
    The client tells the server that it is ready for secure data communication to begin.
14. **ChangeCipherSpec:**
    The server sends a message telling the client that subsequent data will be protected under the newly negotiated CipherSpec and keys and the data is encrypted.
15. **Finished:**
    The server tells the client that it is ready for secure data communication to begin. This is the end of the DTLS handshake.

Handshaking Again (Renegotiation)

Once the initial handshake is finished and application data is flowing, either side is free to initiate a new handshake at any time. An application might like to use a stronger cipher suite for especially critical operations, or a server application might want to require client authentication.

Regardless of the reason, the new handshake takes place over the existing encrypted session, and application data and handshake messages are interleaved until a new session is established.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Your application can initiate a new handshake by using the `SSLEngine.beginHandshake()` method.

Copyright © 1993, 2026, Oracle and/or its affiliates.

9

Java PKI Programmer's Guide

The Java Certification Path API consists of classes and interfaces for handling certification paths, which are also called certification chains. If a certification path meets certain validation rules, it may be used to securely establish the mapping of a public key to a subject.

Topics

PKI Programmer's Guide Overview

Core Classes and Interfaces

Implementing a Service Provider

Appendix A: Standard Names

Appendix B: The CertPath Implementation in the SUN Provider

Appendix C: OCSP Support

Appendix D: CertPath Implementation in JdkLDAP Provider

Appendix E: Disabling Cryptographic Algorithms

PKI Programmer's Guide Overview

The Java Certification Path API defines interfaces and abstract classes for creating, building, and validating certification paths. Implementations may be plugged in using a provider-based interface.

This API is based on the Cryptographic Service Providers architecture, described in the *Java Cryptography Architecture Reference Guide*, and includes algorithm-specific classes for building and validating X.509 certification paths according to the PKIX standards. The PKIX standards were developed by the IETF PKIX working group.

This API was originally specified using the Java Community Process program as Java Specification Request (JSR) 000055. The API was included in the Java SDK, starting with Java SE Development Kit (JDK) 1.4. See JSR 55: Certification Path API.

Who Should Read This Document

This document is intended for two types of experienced developers:

1. Those who want to design secure applications that build or validate certification paths.
2. Those who want to write a service provider implementation for building or validating certification paths.

This document assumes that you have already read Cryptographic Service Providers.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Introduction to Public Key Certificates

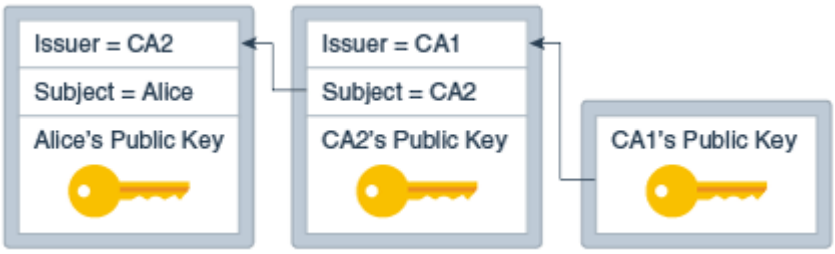
Users of public key applications and systems must be confident that a subject's public key is genuine, i.e., that the associated private key is owned by the subject. Public key certificates are used to establish this trust.

A **public key (or identity) certificate** is a binding of a public key to an identity, which is digitally signed by the private key of another entity, often called a **Certification Authority** (CA). For the remainder of this section, the term CA is used to refer to an entity that signs a certificate.

If the user does not have a trusted copy of the public key of the CA that signed the subject's public key certificate, then another public key certificate vouching for the signing CA is required. This logic can be applied recursively, until a chain of certificates (or **a certification path**) is discovered from a **trust anchor** or a **most-trusted CA** to the target subject (commonly referred to as the **end-entity**). The **most-trusted CA** is usually specified by a certificate issued to a CA that the user directly trusts. In general, a certification path is an ordered list of certificates, usually comprised of the end-entity's public key certificate and zero or more additional certificates. A certification path typically has one or more encodings, allowing it to be safely transmitted across networks and to different operating system architectures.

The following figure illustrates a certification path from a most-trusted CA's public key (CA 1) to the target subject (Alice). The certification path establishes trust in Alice's public key through an intermediate CA named CA2.

Figure 9-1 Certification Path from CA's Public Key (CA 1) to the Target Subject



A certification path must be validated before it can be relied on to establish trust in a subject's public key. Validation can consist of various checks on the certificates contained in the certification path, such as verifying the signatures and checking that each certificate has not been revoked. The PKIX standards define an algorithm for validating certification paths consisting of X.509 certificates.

Often a user may not have a certification path from a most-trusted CA to the subject. Providing services to build or discover certification paths is an important feature of public key enabled systems. RFC 2587 defines an LDAP (Lightweight Directory Access Protocol) schema definition that facilitates the discovery of X.509 certification paths using the LDAP directory service protocol.

Building and validating certification paths is an important part of many standard security protocols such as SSL/TLS/DTLS, S/MIME, and IPsec. The Java Certification Path API provides a set of classes and interfaces for developers who need to integrate this functionality into their applications. This API benefits two types of developers: those who need to write service provider implementations for a specific certification path building or validation

Copyright © 1993, 2026, Oracle and/or its affiliates.

algorithm; and those who need to access standard algorithms for creating, building, and validating certification paths in an implementation-independent manner.

X.509 Certificates and Certificate Revocation Lists (CRLs)

A public-key certificate is a digitally signed statement from one entity saying that the public key and some other information of another entity has some specific value.

The following table defines some of the key terms:

Public Keys
These are numbers associated with a particular entity, and are intended to be known to everyone who needs to have trusted interactions with that entity. Public keys are used to verify signatures.

Digitally Signed
If some data is *digitally signed*, it has been stored with the "identity" of an entity, and a signature that proves that entity knows about the data. The data is rendered unforgeable by signing with the entity's private key.

Identity
A known way of addressing an entity. In some systems the identity is the public key, in others it can be anything from a UNIX UID to an Email address to an X.509 Distinguished Name.

Signature
A signature is computed over some data using the private key of an entity (the signer).

Private Keys
These are numbers, each of which is supposed to be known only to the particular entity whose private key it is (that is, it's supposed to be kept secret). Private and public keys exist in pairs in all public key cryptography systems (also referred to as "public key crypto systems"). In a typical public key crypto system, such as DSA, a private key corresponds to exactly one public key. Private keys are used to compute signatures.

Entity
An entity is a person, organization, program, computer, business, bank, or something else you are trusting to some degree.

Basically, public key cryptography requires access to users' public keys. In a large-scale networked environment it is impossible to guarantee that prior relationships between communicating entities have been established or that a trusted repository exists with all used public keys. Certificates were invented as a solution to this public key distribution problem. Now a *Certification Authority* (CA) can act as a *Trusted Third Party*. CAs are entities (for example, businesses) that are trusted to sign (issue) certificates for other entities. It is assumed that CAs will only create valid and reliable certificates as they are bound by legal agreements. There are many public Certification Authorities, such as Comodo, DigiCert, and GoDaddy.

What Applications use Certificates?

Probably the most widely visible application of X.509 certificates today is in web browsers (such as Mozilla Firefox and Microsoft Internet Explorer) that support the TLS protocol. TLS (Transport Layer Security) is a security protocol that provides privacy and authentication for your network traffic. These browsers can only use this protocol with web servers that support TLS.

Other technologies that rely on X.509 certificates include:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Various code-signing schemes, such as signed Java ARchives, and Microsoft Authenticode.
- Various secure E-Mail standards, such as PEM and S/MIME.

How do I Get a Certificate?

There are two basic techniques used to get certificates:

- You can create one yourself (using the right tools, such as `keytool`).
- You can ask a Certification Authority to issue you one (either directly or using a tool such as `keytool` to generate the request).

The main inputs to the certificate creation process are:

- Matched *public and private keys*, generated using some special tools (such as `keytool`), or a browser. **Only the public key is ever shown to anyone else.** The private key is used to sign data; if someone knows your private key, they can masquerade as you ... perhaps forging legal documents attributed to you!
- You need to provide *information about the entity being certified* (for example, you). This normally includes information such as your name and organizational address. If you ask a CA to issue a certificate for you, you will normally need to provide proof to show correctness of the information.

If you are asking a CA to issue you a certificate, you provide your public key and some information about you. You'll use a tool (such as `keytool` or a browser that supports Certificate Signing Request generation). to digitally sign this information, and send it to the CA. The CA will then generate the certificate and return it.

If you're generating the certificate yourself, you'll take that same information, add a little more (dates during which the certificate is valid, a serial number), and just create the certificate using some tool (such as `keytool`). Not everyone will accept self-signed certificates; one part of the value provided by a CA is to serve as a neutral and trusted introduction service, based in part on their verification requirements, which are openly published in their Certification Service Practices (CSP).

What's Inside an X.509 Certificate?

The X.509 standard defines what information can go into a certificate, and describes how to write it down (the data format). All X.509 certificates have the following data, in addition to the signature:

Version
This identifies which version of the X.509 standard applies to this certificate, which affects what information can be specified in it. Thus far, three versions are defined.

Serial Number
The entity that created the certificate is responsible for assigning it a serial number to distinguish it from other certificates it issues. This information is used in numerous ways, for example when a certificate is revoked its serial number is placed in a Certificate Revocation List (CRL).

Signature Algorithm Identifier
This identifies the algorithm used by the CA to sign the certificate.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Issuer Name
The X.500 name of the entity that signed the certificate. This is normally a CA. Using this certificate implies trusting the entity that signed this certificate. (Note that in some cases, such as *root or top-level* CA certificates, the issuer signs its own certificate.)

Validity Period
Each certificate is valid only for a limited amount of time. This period is described by a start date and time and an end date and time, and can be as short as a few seconds or almost as long as a century. The validity period chosen depends on a number of factors, such as the strength of the private key used to sign the certificate or the amount one is willing to pay for a certificate. This is the expected period that entities can rely on the public value, if the associated private key has not been compromised.

Subject Name
The name of the entity whose public key the certificate identifies. This name uses the X.500 standard, so it is intended to be unique across the Internet. This is the Distinguished Name (DN) of the entity, for example,

```
CN=Java Duke, OU=Java Software Division, O=Sun Microsystems Inc, C=US
```

(These refer to the subject's Common Name, Organizational Unit, Organization, and Country.)

Subject Public Key Information
This is the public key of the entity being named, together with an algorithm identifier which specifies which public key crypto system this key belongs to and any associated key parameters.

X.509 Version 1 has been available since 1988, is widely deployed, and is the most generic.

X.509 Version 2 introduced the concept of subject and issuer unique identifiers to handle the possibility of reuse of subject and/or issuer names over time. Most certificate profile documents strongly recommend that names not be reused, and that certificates should not make use of unique identifiers. Version 2 certificates are not widely used.

X.509 Version 3 is the most recent (1996) and supports the notion of extensions, whereby anyone can define an extension and include it in the certificate. Some common extensions in use today are: *KeyUsage* (limits the use of the keys to particular purposes such as "signing-only") and *AlternativeNames* (allows other identities to also be associated with this public key, for example, DNS names, Email addresses, IP addresses). Extensions can be marked *critical* to indicate that the extension should be checked and enforced/used. For example, if a certificate has the KeyUsage extension marked critical and set to "keyCertSign" then if this certificate is presented during SSL communication, it should be rejected, as the certificate extension indicates that the associated private key should only be used for signing certificates and not for SSL use.

All the data in a certificate is encoded using two related standards called ASN.1/DER. *Abstract Syntax Notation 1* describes data. The *Distinguished Encoding Rules* describe a single way to store and transfer that data.

What Java API Can Be Used to Access and Manage Certificates?

The Certificate API, found in the `java.security.cert` package, includes the following:

- **CertificateFactory** class defines the functionality of a certificate factory, which is used to generate certificate, certificate revocation list (CRL), and certification path objects from their encoding.

Copyright © 1993, 2026, Oracle and/or its affiliates.

- **Certificate** class is an abstract class for managing a variety of certificates. It is an abstraction for certificates that have different formats but important common uses. For example, different types of certificates, such as X.509 and PGP, share general certificate functionality (like encoding and verifying) and some types of information like public key.
- **CRL** class is an abstract class for managing a variety of Certificate Revocation Lists (CRLs).
- **X509Certificate** class is an abstract class for X.509 Certificates. It provides a standard way to access all the attributes of an X.509 certificate.
- **X509Extension** interface is an interface for an X.509 extension. The extensions defined for X.509 v3 certificates and v2 CRLs (Certificate Revocation Lists) provide mechanisms for associating additional attributes with users or public keys, such as for managing the certification hierarchy, and for managing CRL distribution.
- **X509CRL** class is an abstract class for an X.509 Certificate Revocation List (CRL). A CRL is a time-stamped list identifying revoked certificates. It is signed by a Certification Authority (CA) and made freely available in a public repository.
- **X509CRLEntry** class is an abstract class for a CRL entry.

What Java Tool Can Generate, Display, Import, and Export X.509 Certificates?

There is a tool named `keytool` that can be used to create public/private key pairs and X.509 v3 certificates, and to manage keystores. Keys and certificates are used to digitally sign your Java applications and applets (see `jarsigner`).

A *keystore* is a protected database that holds keys and certificates. Access to a keystore is guarded by a password (defined at the time the keystore is created, by the person who creates the keystore, and changeable only when providing the current password). In addition, each private key in a keystore can be guarded by its own password.

Using `keytool`, it is possible to display, import, and export X.509 v1, v2, and v3 certificates stored as files, and to generate new v3 certificates. For examples, see `keytool` in the *Java Development Kit Tool Specifications*.

Core Classes and Interfaces

The core classes of the Java Certification Path API consist of interfaces and classes that support certification path functionality in an algorithm and implementation-independent manner.

The API builds on and extends the existing `java.security.cert` package for handling certificates. The core classes can be broken up into 4 class categories: Basic, Validation, Building, and Storage:

- Basic Certification Path Classes
  - `CertPath`, `CertificateFactory`, and `CertPathParameters`
- Certification Path Validation Classes
  - `CertPathValidator`, `CertPathValidatorResult`, and `CertPathChecker`
- Certification Path Building Classes
  - `CertPathBuilder`, and `CertPathBuilderResult`
- Certificate/CRL Storage Classes
  - `CertStore`, `CertStoreParameters`, `CertSelector`, and `CRLSelector`

Copyright © 1993, 2026, Oracle and/or its affiliates.

The Java Certification Path API also includes a set of algorithm-specific classes modeled for use with the PKIX certification path validation algorithm defined in RFC 5280: *Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. The PKIX Classes are:

- `TrustAnchor`
- `PKIXParameters`
- `PKIXCertPathValidatorResult`
- `PKIXBuilderParameters`
- `PKIXCertPathBuilderResult`
- `PKIXCertPathChecker`
- `PKIXRevocationChecker`

The complete reference documentation for the relevant Certification Path API classes can be found in `java.security.cert` .

Most of the classes and interfaces in the `CertPath` API are not thread-safe. However, there are some exceptions, which will be noted in this guide and in the API specification. Multiple threads that need to access a single non-thread-safe object concurrently should synchronize amongst themselves and provide the necessary locking. Multiple threads each manipulating separate objects need not synchronize.

Topics

Basic Certification Path Classes

Certification Path Validation Classes

Certification Path Building Classes

Certificate/CRL Storage Classes

PKIX Classes

Basic Certification Path Classes

The basic certification path classes provide fundamental functionality for encoding and representing certification paths. The key basic class in the Java Certification Path API is `CertPath`, which encapsulates the universal aspects shared by all types of certification paths. An application uses an instance of the `CertificateFactory` class to create a `CertPath` object.

Topics

The CertPath Class

The CertificateFactory Class

The CertPathParameters Interface

The CertPath Class

The `CertPath` class is an abstract class for certification paths. It defines the functionality shared by all certification path objects. Various certification path types can be implemented by subclassing the `CertPath` class, even though they may have different contents and ordering schemes.

Copyright © 1993, 2026, Oracle and/or its affiliates.

All `CertPath` objects are serializable, immutable and thread-safe and share the following characteristics:

- A type

  This corresponds to the type of the certificates in the certification path, for example: X.509. The type of a `CertPath` is obtained using the method:

  ```
  public String getType()
  ```

  For standard certificate types, see CertificateFactory Types.

- A list of certificates

  The `getCertificates` method returns the list of certificates in the certification path:

  ```
  public abstract List<? extends Certificate> getCertificates()
  ```

  This method returns a `List` of zero or more `java.security.cert.Certificate` objects. The returned `List` and the `Certificates` contained within it are immutable, in order to protect the contents of the `CertPath` object. The ordering of the certificates returned depends on the type. By convention, the certificates in a `CertPath` object of type X.509 are ordered starting with the target certificate and ending with a certificate issued by the trust anchor. That is, the issuer of one certificate is the subject of the following one. The certificate representing the `TrustAnchor` should not be included in the certification path. Unvalidated X.509 `CertPaths` may not follow this convention. PKIX CertPathValidators will detect any departure from these conventions that cause the certification path to be invalid and throw a `CertPathValidatorException`.

- One or more encodings

  Each `CertPath` object supports one or more encodings. These are external encoded forms for the certification path, used when a standard representation of the path is needed outside the Java Virtual Machine (as when transmitting the path over a network to some other party). Each path can be encoded in a default format, the bytes of which are returned using the method:

  ```
  public abstract byte[] getEncoded()
  ```

  Alternatively, the `getEncoded(String)` method returns a specific supported encoding by specifying the encoding format as a `String` (ex: "PKCS7"). For standard encoding formats, see CertPath Encodings.

  ```
  public abstract byte[] getEncoded(String encoding)
  ```

  Also, the `getEncodings` method returns an iterator over the supported encoding format `Strings` (the default encoding format is returned first):

  ```
  public abstract Iterator<String> getEncodings()
  ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

All CertPath objects are also Serializable. CertPath objects are resolved into an alternate CertPath.CertPathRep object during serialization. This allows a CertPath object to be serialized into an equivalent representation regardless of its underlying implementation.

CertPath objects are generated from an encoded byte array or list of Certificates using a CertificateFactory. Alternatively, a CertPathBuilder may be used to try to find a CertPath from a most-trusted CA to a particular subject. Once a CertPath object has been created, it may be validated by passing it to the validate method of CertPathValidator. Each of these concepts are explained in more detail in subsequent sections.

The CertificateFactory Class

The CertificateFactory class is an engine class that defines the functionality of a certificate factory. It is used to generate Certificate, CRL, and CertPath objects.

A CertificateFactory should not be confused with a CertPathBuilder. A CertPathBuilder (discussed later) is used to discover or find a certification path when one does not exist. In contrast, a CertificateFactory is used when a certification path has already been discovered and the caller needs to instantiate a CertPath object from its contents, which exist in a different form such as an encoded byte array or an array of Certificates.

Creating a CertificateFactory Object

See the CertificateFactory section in the *Java Cryptography Architecture Reference Guide* for the details of creating a CertificateFactory object.

Generating CertPath Objects

A CertificateFactory instance generates CertPath objects from a List of Certificate objects or from an InputStream that contains the encoded form of a CertPath. Just like a CertPath, each CertificateFactory supports a default encoding format for certification paths (ex: PKCS#7). To generate a CertPath object and initialize it with the data read from an input stream (in the default encoding format), use the generateCertPath method:

```
public final CertPath generateCertPath(InputStream inStream)
```

or from a particular encoding format:

```
    public final CertPath generateCertPath(InputStream inStream,
                                           String encoding)
```

To find out what encoding formats are supported, use the getCertPathEncodings method (the default encoding is returned first):

```
public final Iterator<String> getCertPathEncodings()
```

To generate a certification path object from a List of Certificate objects, use the following method:

```
public final CertPath generateCertPath(List<? extends Certificate>
certificates)
```

A CertificateFactory always returns CertPath objects that consist of Certificates that are of the same type as the factory. For example, a CertificateFactory of type X.509 returns

Copyright © 1993, 2026, Oracle and/or its affiliates.

`CertPath` objects consisting of certificates that are an instance of `java.security.cert.X509Certificate`.

The following code sample illustrates generating a certification path from a PKCS#7 encoded certificate reply stored in a file:

```
// open an input stream to the file
FileInputStream fis = new FileInputStream(filename);
// instantiate a CertificateFactory for X.509
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// extract the certification path from
// the PKCS7 SignedData structure
CertPath cp = cf.generateCertPath(fis, "PKCS7");
// print each certificate in the path
List<Certificate> certs = cp.getCertificates();
for (Certificate cert : certs) {
    System.out.println(cert);
}
```

Here's another code sample that fetches a certificate chain from a `KeyStore` and converts it to a `CertPath` using a `CertificateFactory`:

```
// instantiate a KeyStore with type PKCS12
KeyStore ks = KeyStore.getInstance("PKCS12");
// load the contents of the KeyStore
ks.load(new FileInputStream("./keystore"),
    "password".toCharArray());
// fetch certificate chain stored with alias "sean"
Certificate[] certArray = ks.getCertificateChain("sean");
// convert chain to a List
List certList = Arrays.asList(certArray);
// instantiate a CertificateFactory for X.509
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// extract the certification path from
// the List of Certificates
CertPath cp = cf.generateCertPath(certList);
```

Note that there is an existing method in `CertificateFactory` named `generateCertificates` that parses a sequence of `Certificates`. For encodings consisting of multiple certificates, use `generateCertificates` when you want to parse a collection of possibly unrelated certificates. Otherwise, use `generateCertPath` when you want to generate a `CertPath` and subsequently validate it with a `CertPathValidator` (discussed later).

The CertPathParameters Interface

The `CertPathParameters` interface is a transparent representation of the set of parameters used with a particular certification path builder or validation algorithm.

Its main purpose is to group (and provide type safety for) all certification path parameter specifications. The `CertPathParameters` interface extends the `Cloneable` interface and defines a `clone()` method that does not throw an exception. All concrete implementations of this interface should implement and override the `Object.clone()` method, if necessary. This allows applications to clone any `CertPathParameters` object.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Objects implementing the `CertPathParameters` interface are passed as arguments to methods of the CertPathValidator and CertPathBuilder classes. Typically, a concrete implementation of the `CertPathParameters` interface will hold a set of input parameters specific to a particular certification path build or validation algorithm. For example, the `PKIXParameters` class is an implementation of the `CertPathParameters` interface that holds a set of input parameters for the PKIX certification path validation algorithm. One such parameter is the set of most-trusted CAs that the caller trusts for anchoring the validation process. This parameter among others is discussed in more detail in the section discussing the PKIXParameters class.

Certification Path Validation Classes

The Java Certification Path API includes classes and interfaces for validating certification paths. An application uses an instance of the `CertPathValidator` class to validate a `CertPath` object. If successful, the result of the validation algorithm is returned in an object implementing the `CertPathValidatorResult` interface.

Topics

The CertPathValidator Class

The CertPathValidatorResult Interface

The CertPathValidator Class

The `CertPathValidator` class is an engine class used to validate a certification path.

Creating a CertPathValidator Object

As with all engine classes, the way to get a `CertPathValidator` object for a particular validation algorithm is to call one of the `getInstance` static factory methods on the `CertPathValidator` class:

```
public static CertPathValidator getInstance(String algorithm)
public static CertPathValidator getInstance(String algorithm,
                                            String provider)
public static CertPathValidator getInstance(String algorithm,
                                            Provider provider)
```

The `algorithm` parameter is the name of a certification path validation algorithm (for example, "PKIX"). Standard `CertPathValidator` algorithm names are listed in the Java Security Standard Algorithm Names.

Validating a Certification Path

Once a `CertPathValidator` object is created, paths can be validated by calling the `validate` method, passing it the certification path to be validated and a set of algorithm-specific parameters:

```
public final CertPathValidatorResult
        validate(CertPath certPath, CertPathParameters params)
        throws CertPathValidatorException,
               InvalidAlgorithmParameterException
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

If the validation algorithm is successful, the result is returned in an object implementing the CertPathValidatorResult interface. Otherwise, a `CertPathValidatorException` is thrown. The `CertPathValidatorException` contains methods that return the `CertPath`, and if relevant, the index of the certificate that caused the algorithm to fail and the root exception or cause of the failure.

Note that the `CertPath` and `CertPathParameters` passed to the `validate` method must be of a type that is supported by the validation algorithm. Otherwise, an `InvalidAlgorithmParameterException` is thrown. For example, a `CertPathValidator` instance that implements the PKIX algorithm validates `CertPath` objects of type X.509 and `CertPathParameters` that are an instance of `PKIXParameters`.

The CertPathValidatorResult Interface

The `CertPathValidatorResult` interface is a transparent representation of the successful result or output of a certification path validation algorithm.

The main purpose of this interface is to group and provide type safety for all validation results. Similar to the `CertPathParameters` interface, `CertPathValidatorResult` extends `Cloneable` and defines a `clone()` method that does not throw an exception. This allows applications to clone any `CertPathValidatorResult` object.

Objects implementing the `CertPathValidatorResult` interface are returned by the `validate` method of CertPathValidatorResult interface when successful. If not successful, a `CertPathValidatorException` is thrown with a description of the failure. Typically, a concrete implementation of the `CertPathValidatorResult` interface will hold a set of output parameters specific to a particular certification path validation algorithm. For example, the `PKIXCertPathValidatorResult` class is an implementation of the `CertPathValidatorResult` interface, which contains methods to get the output parameters of the PKIX certification path validation algorithm. One such parameter is the valid policy tree. This parameter among others is discussed in more detail in the section discussing the PKIXCertPathValidatorResult class.

The following code sample shows how to create a `CertPathValidator` and use it to validate a certification path. The sample assumes that the `CertPath` and `CertPathParameters` objects which are passed to the `validate` method have been previously created; a more complete example will be illustrated in the section describing the PKIX classes.

```
// create CertPathValidator that implements the "PKIX" algorithm
CertPathValidator cpv = null;
try {
    cpv = CertPathValidator.getInstance("PKIX");
} catch (NoSuchAlgorithmException nsae) {
    System.err.println(nsae);
    System.exit(1);
}
// validate certification path ("cp") with specified parameters ("params")
try {
    CertPathValidatorResult cpvResult = cpv.validate(cp, params);
} catch (InvalidAlgorithmParameterException iape) {
    System.err.println("validation failed: " + iape);
    System.exit(1);
} catch (CertPathValidatorException cpve) {
    System.err.println("validation failed: " + cpve);
    System.err.println("index of certificate that caused exception: "
            + cpve.getIndex());
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            System.exit(1);
        }
```

Certification Path Building Classes

The Java Certification Path API includes classes for building (or discovering) certification paths. An application uses an instance of the `CertPathBuilder` class to build a `CertPath` object. If successful, the result of the build is returned in an object implementing the `CertPathBuilderResult` interface.

Topics

The CertPathBuilder Class

The CertPathBuilderResult Interface

The CertPathBuilder Class

The `CertPathBuilder` class is an engine class used to build a certification path.

Creating a CertPathBuilder Object

As with all engine classes, the way to get a `CertPathBuilder` object for a particular build algorithm is to call one of the `getInstance` static factory method on the `CertPathBuilder` class:

```
public static CertPathBuilder getInstance(String algorithm)
public static CertPathBuilder getInstance(String algorithm,
                                          String provider)
public static CertPathBuilder getInstance(String algorithm,
                                          Provider provider)
```

The `algorithm` parameter is the name of a certification path builder algorithm (for example, "PKIX"). Standard `CertPathBuilder` algorithm names are listed in Java Security Standard Algorithm Names.

Building a Certification Path

Once a `CertPathBuilder` object is created, paths can be constructed by calling the `build` method, passing it an algorithm-specific parameter specification:

```
public final CertPathBuilderResult build(CertPathParameters params)
        throws CertPathBuilderException,
               InvalidAlgorithmParameterException
```

If the build algorithm is successful, the result is returned in an object implementing the CertPathBuilderResult interface. Otherwise, a `CertPathBuilderException` is thrown containing information about the failure; for example, the underlying exception (if any) and an error message.

Note that the `CertPathParameters` passed to the `build` method must be of a type that is supported by the build algorithm. Otherwise, an `InvalidAlgorithmParameterException` is thrown.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The CertPathBuilderResult Interface

The `CertPathBuilderResult` interface is a transparent representation of the result or output of a certification path builder algorithm.

This interface contains a method to return the certification path that has been successfully built:

```
public CertPath getCertPath()
```

The purpose of the `CertPathBuilderResult` interface is to group (and provide type safety for) all build results. Like the `CertPathValidatorResult` interface, `CertPathBuilderResult` extends `Cloneable` and defines a `clone()` method that does not throw an exception. This allows applications to clone any `CertPathBuilderResult` object.

Objects implementing the `CertPathBuilderResult` interface are returned by the `build` method of CertPathBuilder.

The following code sample shows how to create a `CertPathBuilder` and use it to build a certification path. The sample assumes that the `CertPathParameters` object which is passed to the `build` method has been previously created; a more complete example will be illustrated in the section describing the PKIX classes.

```
// create CertPathBuilder that implements the "PKIX" algorithm
CertPathBuilder cpb = null;
try {
    cpb = CertPathBuilder.getInstance("PKIX");
} catch (NoSuchAlgorithmException nsae) {
    System.err.println(nsae);
    System.exit(1);
}
// build certification path using specified parameters ("params")
try {
    CertPathBuilderResult cpbResult = cpb.build(params);
    CertPath cp = cpbResult.getCertPath();
    System.out.println("build passed, path contents: " + cp);
} catch (InvalidAlgorithmParameterException iape) {
    System.err.println("build failed: " + iape);
    System.exit(1);
} catch (CertPathBuilderException cpbe) {
    System.err.println("build failed: " + cpbe);
    System.exit(1);
}
```

Certificate/CRL Storage Classes

The Java Certification Path API includes the `CertStore` class for retrieving certificates and CRLs from a repository.

This class enables a caller to specify the repository a `CertPathValidator` or `CertPathBuilder` implementation should use to find certificates and CRLs. See the `addCertStores` method of the PKIXParameters class.

Copyright © 1993, 2026, Oracle and/or its affiliates.

A `CertPathValidator` implementation may use the `CertStore` object that the caller specifies as a callback mechanism to fetch CRLs for performing revocation checks. Similarly, a `CertPathBuilder` may use the `CertStore` as a callback mechanism to fetch certificates and, if performing revocation checks, CRLs.

Topics

The CertStore Class

The CertStoreParameters Interface

The CertSelector and CRLSelector Interfaces

The CertStore Class

The `CertStore` class is an engine class used to provide the functionality of a certificate and certificate revocation list (CRL) repository.

This class can be used by CertPathBuilder and CertPathValidator implementations to find certificates and CRLs, or as a general purpose certificate and CRL retrieval mechanism.

Unlike the `java.security.KeyStore` class, which provides access to a cache of private keys and trusted certificates, a `CertStore` is designed to provide access to a potentially vast repository of untrusted certificates and CRLs. For example, an LDAP implementation of `CertStore` provides access to certificates and CRLs stored in one or more directories using the LDAP protocol.

All public methods of `CertStore` objects are thread-safe. That is, multiple threads may concurrently invoke these methods on a single `CertStore` object (or more than one) with no ill effects. This allows a `CertPathBuilder` to search for a CRL while simultaneously searching for further certificates, for instance.

Creating a CertStore Object

As with all engine classes, the way to get a `CertStore` object for a particular repository type is to call one of the `getInstance` static factory methods on the `CertStore` class:

```
public static CertStore getInstance(String type,
        CertStoreParameters params)
public static CertStore getInstance(String type,
        CertStoreParameters params, String provider)
public static CertStore getInstance(String type,
        CertStoreParameters params, Provider provider)
```

The `type` parameter is the name of a certificate repository type (for example, "LDAP"). Standard `CertStore` types are listed in Java Security Standard Algorithm Names.

The initialization parameters (`params`) are specific to the repository type. For example, the initialization parameters for a server-based repository may include the hostname and the port of the server. An `InvalidAlgorithmParameterException` is thrown if the parameters are invalid for this `CertStore` type. The `getCertStoreParameters` method returns the `CertStoreParameters` that were used to initialize a `CertStore`:

```
public final CertStoreParameters getCertStoreParameters()
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Retrieving Certificates

After you have created a `CertStore` object, you can retrieve certificates from the repository using the `getCertificates` method. This method takes a CertSelector (discussed in more detail later) object as an argument, which specifies a set of selection criteria for determining which certificates should be returned:

```
        public final Collection<? extends Certificate>
getCertificates(CertSelector selector)
                throws CertStoreException
```

This method returns a `Collection` of `java.security.cert.Certificate` objects that satisfy the selection criteria. An empty `Collection` is returned if there are no matches. A `CertStoreException` is usually thrown if an unexpected error condition is encountered, such as a communications failure with a remote repository.

For some `CertStore` implementations, it may not be feasible to search the entire repository for certificates or CRLs that match the specified selection criteria. In these instances, the `CertStore` implementation may use information that is specified in the selectors to locate certificates and CRLs. For instance, an LDAP `CertStore` may not search all entries in the directory. Instead, it may just search entries that are likely to contain the certificates it is looking for. If the `CertSelector` provided does not provide enough information for the LDAP `CertStore` to determine which entries it should look in, the LDAP `CertStore` may throw a `CertStoreException`.

Retrieving CRLs

You can also retrieve CRLs from the repository using the `getCRLs` method. This method takes a CRLSelector (discussed in more detail later) object as an argument, which specifies a set of selection criteria for determining which CRLs should be returned:

```
        public final Collection<? extends CRL> getCRLs(CRLSelector selector)
                throws CertStoreException
```

This method returns a `Collection` of `java.security.cert.CRL` objects that satisfy the selection criteria. An empty `Collection` is returned if there are no matches.

The CertStoreParameters Interface

The `CertStoreParameters` interface is a transparent representation of the set of parameters used with a particular `CertStore`.

The main purpose of this interface is to group and provide type safety for all certificate storage parameter specifications. The `CertStoreParameters` interface extends the `Cloneable` interface and defines a `clone` method that does not throw an exception. Implementations of this interface should implement and override the `Object.clone()` method, if necessary. This allows applications to clone any `CertStoreParameters` object.

Objects implementing the `CertStoreParameters` interface are passed as arguments to the `getInstance` method of the `CertStore` class. Two classes implementing the `CertStoreParameters` interface are defined in this API: the LDAPCertStoreParameters class and the CollectionCertStoreParameters class.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The LDAPCertStoreParameters Class

The `LDAPCertStoreParameters` class is an implementation of the `CertStoreParameters` interface and holds a set of minimal initialization parameters (host and port number of the directory server) for retrieving certificates and CRLs from a `CertStore` of type **LDAP**.

See `LDAPCertStoreParameters`.

The CollectionCertStoreParameters Class

The `CollectionCertStoreParameters` class is an implementation of the `CertStoreParameters` interface and holds a set of initialization parameters for retrieving certificates and CRLs from a `CertStore` of type **Collection**.

See `CollectionCertStoreParameters`.

The CertSelector and CRLSelector Interfaces

The `CertSelector` and `CRLSelector` interfaces are a specification of the set of criteria for selecting certificates and CRLs from a collection or large group of certificates and CRLs.

The interfaces group and provide type safety for all selector specifications. Each selector interface extends `Cloneable` and defines a `clone()` method that does not throw an exception. This allows applications to clone any `CertSelector` or `CRLSelector` object.

The `CertSelector` and `CRLSelector` interfaces each define a method named `match`. The `match` method takes a `Certificate` or `CRL` object as an argument and returns `true` if the object satisfies the selection criteria. Otherwise, it returns `false`. The `match` method for the `CertSelector` interface is defined as follows:

```
public boolean match(Certificate cert)
```

and for the `CRLSelector` interface:

```
public boolean match(CRL crl)
```

Typically, objects implementing these interfaces are passed as parameters to the `getCertificates` and `getCRLs` methods of the `CertStore` class. These methods return a `Collection` of `Certificates` or `CRLs` from the `CertStore` repository that match the specified selection criteria. `CertSelectors` may also be used to specify the validation constraints on a target or end-entity certificate in a certification path (see for example, the `PKIXParameters.setTargetCertConstraints` method.)

The X509CertSelector Class

The `X509CertSelector` class is an implementation of the `CertSelector` interface that defines a set of criteria for selecting X.509 certificates.

An `X509Certificate` object must match *all* of the specified criteria to be selected by the `match` method. The selection criteria are designed to be used by a `CertPathBuilder` implementation to discover potential certificates as it builds an X.509 certification path.

Copyright © 1993, 2026, Oracle and/or its affiliates.

For example, the `setSubject` method of `X509CertSelector` allows a PKIX `CertPathBuilder` to filter out `X509Certificates` that do not match the issuer name of the preceding `X509Certificate` in a partially completed chain. By setting this and other criteria in an `X509CertSelector` object, a `CertPathBuilder` is able to discard irrelevant certificates and more easily find an X.509 certification path that meets the requirements specified in the `CertPathParameters` object.

See RFC 5280 for definitions of the X.509 certificate extensions mentioned in this section.

Creating an X509CertSelector Object

An `X509CertSelector` object is created by calling the default constructor:

```
public X509CertSelector()
```

No criteria are initially set (any `X509Certificate` will match).

Setting Selection Criteria

The selection criteria allow a caller to match on different components of an X.509 certificate. A few of the methods for setting selection criteria are described here. See `X509CertSelector`.

The `setIssuer` methods set the issuer criterion:

```
public void setIssuer(X500Principal issuer)
public void setIssuer(String issuerDN)
public void setIssuer(byte[] issuerDN)
```

The specified distinguished name (in `X500Principal`, RFC 2253 String or ASN.1 DER encoded form) must match the issuer distinguished name in the certificate. If null, any issuer distinguished name will do. Note that use of an `X500Principal` to represent a distinguished name is preferred because it is more efficient and suitably typed.

Similarly, the `setSubject` methods set the subject criterion:

```
public void setSubject(X500Principal subject)
public void setSubject(String subjectDN)
public void setSubject(byte[] subjectDN)
```

The specified distinguished name (in `X500Principal`, RFC 2253 String or ASN.1 DER encoded form) must match the subject distinguished name in the certificate. If null, any subject distinguished name will do.

The `setSerialNumber` method sets the serialNumber criterion:

```
public void setSerialNumber(BigInteger serial)
```

The specified serial number must match the certificate serial number in the certificate. If null, any certificate serial number will do.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The `setAuthorityKeyIdentifier` method sets the authorityKeyIdentifier criterion:

```
public void setAuthorityKeyIdentifier(byte[] authorityKeyID)
```

The certificate must contain an Authority Key Identifier extension matching the specified value. If null, no check will be done on the authorityKeyIdentifier criterion.

The `setCertificateValid` method sets the certificateValid criterion:

```
public void setCertificateValid(Date certValid)
```

The specified date must fall within the certificate validity period for the certificate. If null, any date is valid.

The `setKeyUsage` method sets the keyUsage criterion:

```
public void setKeyUsage(boolean[] keyUsage)
```

The certificate's Key Usage Extension must allow the specified key usage values (those which are set to true). If null, no keyUsage check will be done.

Getting Selection Criteria

The current values for each of the selection criteria can be retrieved using an appropriate `get` method. See `X509CertSelector` .

Here is an example of retrieving X.509 certificates from an LDAP `CertStore` with the `X509CertSelector` class.

First, we create the `LDAPCertStoreParameters` object that we will use to initialize the `CertStore` object with the hostname and port of the LDAP server:

```
LDAPCertStoreParameters lcsp = new
        LDAPCertStoreParameters("ldap.sun.com", 389);
```

Next, create the CertStore object, and pass it the `LDAPCertStoreParameters` object, as in the following statement:

```
CertStore cs = CertStore.getInstance("LDAP", lcsp);
```

This call creates a `CertStore` object that retrieves certificates and CRLs from an LDAP repository using the schema defined in RFC 2587.

The following block of code establishes an `X509CertSelector` to retrieve all unexpired (as of the current date and time) end-entity certificates issued to a particular subject with 1) a key

Copyright © 1993, 2026, Oracle and/or its affiliates.

usage that allows digital signatures, and 2) a subject alternative name with a specific email address:

```
X509CertSelector xcs = new X509CertSelector();

// select only unexpired certificates
xcs.setCertificateValid(new Date());

// select only certificates issued to
// 'CN=alice, O=xyz, C=us'
xcs.setSubject(new X500Principal("CN=alice, O=xyz, C=us"));

// select only end-entity certificates
xcs.setBasicConstraints(-2);

// select only certificates with a digitalSignature
// keyUsage bit set (set the first entry in the
// boolean array to true)
boolean[] keyUsage = {true};
xcs.setKeyUsage(keyUsage);

// select only certificates with a subjectAltName of
// 'alice@xyz.example.com' (1 is the integer value of
// an RFC822Name)
xcs.addSubjectAlternativeName(1, "alice@xyz.example.com");
```

Then we pass the selector to the `getCertificates` method of our `CertStore` object that we previously created:

```
Collection<Certificate> certs = cs.getCertificates(xcs);
```

A PKIX `CertPathBuilder` may use similar code to help discover and sort through potential certificates by discarding those that do not meet validation constraints or other criteria.

The X509CRLSelector Class

The `X509CRLSelector` class is an implementation of the `CRLSelector` interface that defines a set of criteria for selecting X.509 CRLs.

An `X509CRL` object must match *all* of the specified criteria to be selected by the `match` method. The selection criteria are designed to be useful to a `CertPathValidator` or `CertPathBuilder` implementation that must retrieve CRLs from a repository to check the revocation status of certificates in an X.509 certification path.

For example, the `setDateAndTime` method of `X509CRLSelector` allows a PKIX `CertPathValidator` to filter out `X509CRL`s that have been issued after or expire before the time indicated. By setting this and other criteria in an `X509CRLSelector` object, it allows the `CertPathValidator` to discard irrelevant CRLs and more easily check if a certificate has been revoked.

Please refer to RFC 5280 for definitions of the X.509 CRL fields and extensions mentioned in this section.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Creating an X509CRLSelector Object

An `X509CRLSelector` object is created by calling the default constructor:

```
public X509CRLSelector()
```

No criteria are initially set (any `X509CRL` will match).

Setting Selection Criteria

The selection criteria allow a caller to match on different components of an X.509 CRL. Most of the methods for setting selection criteria are described here. Please refer to the `X509CRLSelector Class` API documentation for details on the remaining methods.

The `setIssuers` and `setIssuerNames` methods set the issuerNames criterion:

```
public void setIssuers(Collection<X500Principal> issuers)
public void setIssuerNames(Collection<?> names)
```

The issuer distinguished name in the CRL must match at least one of the specified distinguished names. The `setIssuers` method is preferred as the use of `X500Principals` to represent distinguished names is more efficient and suitably typed. For the `setIssuerNames` method, each entry of the `names` argument is either a `String` or a byte array (representing the name, in RFC 2253 or ASN.1 DER encoded form, respectively). If null, any issuer distinguished name will do.

The `setMinCRLNumber` and `setMaxCRLNumber` methods set the minCRLNumber and maxCRLNumber criterion:

```
public void setMinCRLNumber(BigInteger minCRL)
public void setMaxCRLNumber(BigInteger maxCRL)
```

The CRL must have a CRL Number extension whose value is greater than or equal to the specified value if the `setMinCRLNumber` method is called, and less than or equal to the specified value if the `setMaxCRLNumber` method is called. If the value passed to one of these methods is null, the corresponding check is not done.

The `setDateAndTime` method sets the dateAndTime criterion:

```
public void setDateAndTime(Date dateAndTime)
```

The specified date must be equal to or later than the value of the thisUpdate component of the CRL and earlier than the value of the nextUpdate component. If null, no dateAndTime check will be done.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The `setCertificateChecking` method sets the certificate whose revocation status is being checked:

```
public void setCertificateChecking(X509Certificate cert)
```

This is not a criterion. Rather, it is optional information that may help a `CertStore` find CRLs that would be relevant when checking revocation for the specified certificate. If null is specified, then no such optional information is provided. An application should always call this method when checking revocation for a particular certificate, as it may provide the `CertStore` with more information for finding the correct CRLs and filtering out irrelevant ones.

Getting Selection Criteria

The current values for each of the selection criteria can be retrieved using an appropriate `get` method. Please refer to the `X509CRLSelector Class` API documentation for further details on these methods.

Creating an `X509CRLSelector` to retrieve CRLs from an LDAP repository is similar to the `X509CertSelector` example. Suppose we want to retrieve all current (as of the current date and time) CRLs issued by a specific CA and with a minimum CRL number. First, we create an `X509CRLSelector` object and call the appropriate methods to set the selection criteria:

```
X509CRLSelector xcrls = new X509CRLSelector();
// select CRLs satisfying current date and time
xcrls.setDateAndTime(new Date());
// select CRLs issued by 'O=xyz, C=us'
xcrls.addIssuerName("O=xyz, C=us");
// select only CRLs with a CRL number at least '2'
xcrls.setMinCRLNumber(new BigInteger("2"));
```

Then we pass the selector to the `getCRLs` method of our `CertStore` object (created in the `X509CertSelector` example):

```
Collection<CRL> crls = cs.getCRLs(xcrls);
```

PKIX Classes

The Java Certification Path API includes a set of algorithm-specific classes modeled for use with the PKIX certification path validation algorithm.

The PKIX certification path validation algorithm is defined in RFC 5280: *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*.

Topics

The TrustAnchor Class

The PKIXParameters Class

The CertPathValidatorResult Interface

The PolicyNode Interface and PolicyQualifierInfo Class

Copyright © 1993, 2026, Oracle and/or its affiliates.

The PKIXBuilderParameters Class

The PKIXCertPathBuilderResult Class

The PKIXCertPathChecker Class

Using PKIXCertPathChecker in Certificate Path Validation

The TrustAnchor Class

The `TrustAnchor` class represents a "most-trusted CA", which is used as a trust anchor for validating X.509 certification paths.

A `TrustAnchor` includes the public key of the CA, the CA's name, and any constraints on the set of paths that can be validated using this key. These parameters can be specified in the form of a trusted `X509Certificate` or as individual parameters.

All `TrustAnchor` objects are immutable and thread-safe. That is, multiple threads may concurrently invoke the methods defined in this class on a single `TrustAnchor` object (or more than one) with no ill effects. Requiring `TrustAnchor` objects to be immutable and thread-safe allows them to be passed around to various pieces of code without worrying about coordinating access.

> **Note**
>
> Although this class is described as a PKIX class it may be used with other X.509 certification path validation algorithms.

Creating a TrustAnchor Object

To instantiate a `TrustAnchor` object, a caller must specify "the most-trusted CA" as a trusted `X509Certificate` or public key and distinguished name pair. The caller may also optionally specify name constraints that are applied to the trust anchor by the validation algorithm during initialization. Note that support for name constraints on trust anchors is not required by the PKIX algorithm, therefore a PKIX `CertPathValidator` or `CertPathBuilder` may choose not to support this parameter and instead throw an exception. Use one of the following constructors to create a `TrustAnchor` object:

```
public TrustAnchor(X509Certificate trustedCert,
        byte[] nameConstraints)
public TrustAnchor(X500Principal caPrincipal, PublicKey pubKey,
        byte[] nameConstraints)
public TrustAnchor(String caName, PublicKey pubKey,
        byte[] nameConstraints)
```

The `nameConstraints` parameter is specified as a byte array containing the ASN.1 DER encoding of a NameConstraints extension. An `IllegalArgumentException` is thrown if the name constraints cannot be decoded (are not formatted correctly).

Copyright © 1993, 2026, Oracle and/or its affiliates.

Getting Parameter Values

Each of the parameters can be retrieved using a corresponding get method:

```
public final X509Certificate getTrustedCert()
public final X500Principal getCA()
public final String getCAName()
public final PublicKey getCAPublicKey()
public final byte[] getNameConstraints()
```

> **Note**
>
> The `getTrustedCert` method returns `null` if the trust anchor was specified as a public key and name pair. Likewise, the `getCA`, `getCAName` and `getCAPublicKey` methods return `null` if the trust anchor was specified as an `X509Certificate`.

The PKIXParameters Class

The `PKIXParametersClass` class specifies the set of input parameters defined by the PKIX certification path validation algorithm. It also includes a few additional useful parameters.

This class implements the CertPathParameters interface.

An X.509 `CertPath` object and a `PKIXParameters` object are passed as arguments to the `validate` method of a CertPathValidator instance implementing the PKIX algorithm. The `CertPathValidator` uses the parameters to initialize the PKIX certification path validation algorithm.

Creating a PKIXParameters Object

To instantiate a `PKIXParameters` object, a caller must specify "the most-trusted CA(s)" as defined by the PKIX validation algorithm. The most-trusted CAs can be specified using one of two constructors:

```
public PKIXParameters(Set<TrustAnchor> trustAnchors)
    throws InvalidAlgorithmParameterException
public PKIXParameters(KeyStore keystore)
    throws KeyStoreException, InvalidAlgorithmParameterException
```

The first constructor allows the caller to specify the most-trusted CAs as a `Set` of `TrustAnchor` objects. Alternatively, a caller can use the second constructor and specify a `KeyStore` instance containing trusted certificate entries, each of which will be considered as a most-trusted CA.

Setting Parameter Values

After a `PKIXParameters` object has been created, a caller can set (or replace the current value of) various parameters. A few of the methods for setting parameters are described here. Please refer to the `PKIXParameters` API documentation for details on the other methods.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The `setInitialPolicies` method sets the initial policy identifiers, as specified by the PKIX validation algorithm. The elements of the `Set` are object identifiers (OIDs) represented as a `String`. If the `initialPolicies` parameter is null or not set, any policy is acceptable:

```
public void setInitialPolicies(Set<String> initialPolicies)
```

The `setDate` method sets the time for which the validity of the path should be determined. If the `date` parameter is not set or is null, the current date is used:

```
public void setDate(Date date)
```

The `setPolicyMappingInhibited` method sets the value of the policy mapping inhibited flag. The default value for the flag, if not specified, is false:

```
public void setPolicyMappingInhibited(boolean val)
```

The `setExplicitPolicyRequired` method sets the value of the explicit policy required flag. The default value for the flag, if not specified, is false:

```
public void setExplicitPolicyRequired(boolean val)
```

The `setAnyPolicyInhibited` method sets the value of the any policy inhibited flag. The default value for the flag, if not specified, is false:

```
public void setAnyPolicyInhibited(boolean val)
```

The `setTargetCertConstraints` method allows the caller to set constraints on the target or end-entity certificate. For example, the caller can specify that the target certificate must contain a specific subject name. The constraints are specified as a `CertSelector` object. If the `selector` parameter is null or not set, no constraints are defined on the target certificate:

```
public void setTargetCertConstraints(CertSelector selector)
```

The `setCertStores` method allows a caller to specify a `List` of `CertStore` objects that will be used by a PKIX implementation of `CertPathValidator` to find CRLs for path validation. This provides an extensible mechanism for specifying where to locate CRLs. The `setCertStores` method takes a `List` of `CertStore` objects as a parameter. The first `CertStores` in the list may be preferred to those that appear later.

```
public void setCertStores(List<CertStore> stores)
```

The `setCertPathCheckers` method allows a caller to extend the PKIX validation algorithm by creating implementation-specific certification path checkers. For example, this mechanism can

Copyright © 1993, 2026, Oracle and/or its affiliates.

be used to process private certificate extensions. The `setCertPathCheckers` method takes a list of PKIXCertPathChecker (discussed later) objects as a parameter:

```
public void setCertPathCheckers(List<PKIXCertPathChecker> checkers)
```

The `setRevocationEnabled` method allows a caller to disable revocation checking. Revocation checking is enabled by default, since it is a required check of the PKIX validation algorithm. However, PKIX does not define how revocation should be checked. An implementation may use CRLs or OCSP, for example. This method allows the caller to disable the implementation's default revocation checking mechanism if it is not appropriate. A different revocation checking mechanism can then be specified by calling the `setCertPathCheckers` method, and passing it a `PKIXCertPathChecker` that implements the alternate mechanism.

```
public void setRevocationEnabled(boolean val)
```

The `setPolicyQualifiersRejected` method allows a caller to enable or disable policy qualifier processing. When a `PKIXParameters` object is created, this flag is set to `true`. This setting reflects the most common (and simplest) strategy for processing policy qualifiers. Applications that want to use a more sophisticated policy must set this flag to `false`.

```
public void setPolicyQualifiersRejected(boolean qualifiersRejected)
```

Getting Parameter Values

The current values for each of the parameters can be retrieved using an appropriate `get` method. Please refer to the `Class PKIXParameters` API documentation for further details on these methods.

The PKIXCertPathValidatorResult Class

The `PKIXCertPathValidatorResult` class represents the result of the PKIX certification path validation algorithm.

This class implements the CertPathValidatorResult interface. It holds the valid policy tree and subject public key resulting from the validation algorithm, and includes methods (`getPolicyTree()` and `getPublicKey()`) for returning them. Instances of `PKIXCertPathValidatorResult` are returned by the `validate` method of CertPathValidator objects implementing the PKIX algorithm.

Please refer to the `PKIXCertPathValidatorResult` API documentation for more detailed information on this class.

The PolicyNode Interface and PolicyQualifierInfo Class

The PKIX validation algorithm defines several outputs related to certificate policy processing. Most applications will not need to use these outputs, but all providers that implement the PKIX validation or building algorithm must support them.

The `PolicyNode` interface represents a node of a valid policy tree resulting from a successful execution of the PKIX certification path validation. An application can obtain the root of a valid

Copyright © 1993, 2026, Oracle and/or its affiliates.

policy tree using the `getPolicyTree` method of `PKIXCertPathValidatorResult`. Policy Trees are discussed in more detail in the RFC 5280.

The `getPolicyQualifiers` method of `PolicyNode` returns a `Set` of `PolicyQualifierInfo` objects, each of which represents a policy qualifier contained in the Certificate Policies extension of the relevant certificate that this policy applies to.

Most applications will not need to examine the valid policy tree and policy qualifiers. They can achieve their policy processing goals by setting the policy-related parameters in `PKIXParameters`. However, the valid policy tree is available for more sophisticated applications, especially those that process policy qualifiers.

Please refer to the `Interface PolicyNode` and `PolicyQualifierInfo` API documentation for more detailed information on these classes.

Example 9-1 Example of Validating a Certification Path using the PKIX algorithm

This is an example of validating a certification path with the PKIX validation algorithm. The example ignores most of the exception handling and assumes that the certification path and public key of the trust anchor have already been created.

First, create the CertPathValidator, as in the following line:

```
CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
```

The next step is to create a TrustAnchor object. This will be used as an anchor for validating the certification path. In this example, the most-trusted CA is specified as a public key and name (name constraints are not applied and are specified as `null`):

```
TrustAnchor anchor = new TrustAnchor("O=xyz,C=us", pubkey, null);
```

The next step is to create a PKIXParameters object. This will be used to populate the parameters used by the PKIX algorithm. In this example, we pass to the constructor a `Set` containing a single element - the `TrustAnchor` we created in the previous step:

```
PKIXParameters params = new PKIXParameters(Collections.singleton(anchor));
```

Next, we populate the parameters object with constraints or other parameters used by the validation algorithm. In this example, we enable the explicitPolicyRequired flag and specify a set of initial policy OIDs (the contents of the set are not shown):

```
// set other PKIX parameters here
params.setExplicitPolicyRequired(true);
params.setInitialPolicies(policyIds);
```

The final step is to validate the certification path using the input parameter set we have created:

```
try {
    PKIXCertPathValidatorResult result =
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            (PKIXCertPathValidatorResult) cpv.validate(certPath, params);
        PolicyNode policyTree = result.getPolicyTree();
        PublicKey subjectPublicKey = result.getPublicKey();
    } catch (CertPathValidatorException cpve) {
        System.out.println("Validation failure, cert["
            + cpve.getIndex() + "] :" + cpve.getMessage());
    }
```

If the validation algorithm is successful, the policy tree and subject public key resulting from the validation algorithm are obtained using the `getPolicyTree` and `getPublicKey` methods of `PKIXCertPathValidatorResult`.

Otherwise, a `CertPathValidatorException` is thrown and the caller can catch the exception and print some details about the failure, such as the error message and the index of the certificate that caused the failure.

The PKIXBuilderParameters Class

The `PKIXBuilderParameters` class specifies the set of parameters to be used with `CertPathBuilder` class.

This class (which extends the PKIXParameters class) specifies the set of parameters to be used with CertPathBuilder class that build certification paths validated against the PKIX certification path validation algorithm.

A `PKIXBuilderParameters` object is passed as an argument to the `build` method of a CertPathBuilder instance implementing the PKIX algorithm. All PKIX `CertPathBuilders` *must* return certification paths which have been validated according to the PKIX certification path validation algorithm.

Please note that the mechanism that a PKIX `CertPathBuilder` uses to validate a constructed path is an implementation detail. For example, an implementation might attempt to first build a path with minimal validation and then fully validate it using an instance of a PKIX `CertPathValidator`, whereas a more efficient implementation may validate more of the path as it is building it, and backtrack to previous stages if it encounters validation failures or dead-ends.

Creating a PKIXBuilderParameters Object

Creating a `PKIXBuilderParameters` object is similar to creating a `PKIXParameters` object. However, a caller *must* specify constraints on the target or end-entity certificate when creating a `PKIXBuilderParameters` object. These constraints should provide the `CertPathBuilder` with enough information to find the target certificate. The constraints are specified as a `CertSelector` object. Use one of the following constructors to create a `PKIXBuilderParameters` object:

```
public PKIXBuilderParameters(Set<TrustAnchor> trustAnchors,
        CertSelector targetConstraints)
        throws InvalidAlgorithmParameterException
public PKIXBuilderParameters(KeyStore keystore,
        CertSelector targetConstraints)
        throws KeyStoreException, InvalidAlgorithmParameterException
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Getting/Setting Parameter Values

The `PKIXBuilderParameters` class inherits all of the parameters that can be set in the `PKIXParameters` class. In addition, the `setMaxPathLength` method can be called to place a limit on the maximum number of certificates in a certification path:

```
public void setMaxPathLength(int maxPathLength)
```

The `maxPathLength` parameter specifies the maximum number of non-self-issued intermediate certificates that may exist in a certification path. A `CertPathBuilder` instance implementing the PKIX algorithm must not build paths longer than the length specified. If the value is 0, the path can only contain a single certificate. If the value is -1, the path length is unconstrained (i.e., there is no maximum). The default maximum path length, if not specified, is 5. This method is useful to prevent the `CertPathBuilder` from spending resources and time constructing long paths that may or may not meet the caller's requirements.

If any of the CA certificates in the path contain a Basic Constraints extension, the value of the pathLenConstraint component of the extension overrides the value of the `maxPathLength` parameter whenever the result is a certification path of smaller length. There is also a corresponding `getMaxPathLength` method for retrieving this parameter:

```
public int getMaxPathLength()
```

Also, the `setCertStores` method (inherited from the `PKIXParameters` class) is typically used by a PKIX implementation of `CertPathBuilder` to find Certificates for path construction as well as finding CRLs for path validation. This provides an extensible mechanism for specifying where to locate Certificates and CRLs.

The PKIXCertPathBuilderResult Class

The `PKIXCertPathBuilderResult` class represents the successful result of the PKIX certification path construction algorithm.

This class extends the `PKIXCertPathValidatorResult` class and implements the CertPathBuilder interface. Instances of `PKIXCertPathBuilderResult` are returned by the `build` method of CertPathBuilder objects implementing the PKIX algorithm.

The `getCertPath` method of a `PKIXCertPathBuilderResult` instance always returns a `CertPath` object validated using the PKIX certification path validation algorithm. The returned `CertPath` object does not include the most-trusted CA certificate that may have been used to anchor the path. Instead, use the `getTrustAnchor` method to get the `Certificate` of the most-trusted CA.

See the `PKIXCertPathBuilderResult` API documentation for more detailed information on this class.

Example 9-2 Example of Building a Certification Path using the PKIX algorithm

This is an example of building a certification path validated against the PKIX algorithm. Some details have been left out, such as exception handling, and the creation of the trust anchors and certificates for populating the `CertStore`.

Copyright © 1993, 2026, Oracle and/or its affiliates.

First, create the CertPathBuilder, as in the following example:

```
CertPathBuilder cpb = CertPathBuilder.getInstance("PKIX");
```

This call creates a `CertPathBuilder` object that returns paths validated against the PKIX algorithm.

The next step is to create a PKIXBuilderParameters object. This will be used to populate the PKIX parameters used by the `CertPathBuilder`:

```
// Create parameters object, passing it a Set of
// trust anchors for anchoring the path
// and a target subject DN.
X509CertSelector targetConstraints = new X509CertSelector();
targetConstraints.setSubject("CN=alice,O=xyz,C=us");
PKIXBuilderParameters params =
    new PKIXBuilderParameters(trustAnchors, targetConstraints);
```

The next step is to specify the `CertStore` that the `CertPathBuilder` will use to look for certificates and CRLs. For this example, we will populate a Collection `CertStore` with the certificates and CRLs:

```
CollectionCertStoreParameters ccsp =
    new CollectionCertStoreParameters(certsAndCrls);
CertStore store = CertStore.getInstance("Collection", ccsp);
params.addCertStore(store);
```

The next step is to build the certification path using the input parameter set we have created:

```
try {
    PKIXCertPathBuilderResult result =
        (PKIXCertPathBuilderResult) cpb.build(params);
    CertPath cp = result.getCertPath();
} catch (CertPathBuilderException cpbe) {
    System.out.println("build failed: " + cpbe.getMessage());
}
```

If the `CertPathBuilder` cannot build a path that meets the supplied parameters it will throw a `CertPathBuilderException`. Otherwise, the validated certification path can be obtained from the `PKIXCertPathBuilderResult` using the `getCertPath` method.

The PKIXCertPathChecker Class

The PKIXCertPathChecker class allows a user to extend a PKIX `CertPathValidator` or `CertPathBuilder` implementation. This is an advanced feature that most users will not need to understand. However, anyone implementing a PKIX service provider should read this section

The `PKIXCertPathChecker` class is an abstract class that executes one or more checks on an X.509 certificate. Developers should create concrete implementations of the

Copyright © 1993, 2026, Oracle and/or its affiliates.

`PKIXCertPathChecker` class when it is necessary to dynamically extend a PKIX `CertPathValidator` or `CertPathBuilder` implementation at runtime. The following are a few examples of when a `PKIXCertPathChecker` implementation is useful:

- If the revocation mechanism supplied by a PKIX `CertPathValidator` or `CertPathBuilder` implementation is not adequate: For example, you can use the `PKIXRevocationChecker` (introduced in JDK 8; see Check Revocation Status of Certificates with PKIXRevocationChecker Class) to have more control over the revocation mechanism, or you can implement your own `PKIXCertPathChecker` to check that certificates have not been revoked.
- If the user wants to recognize certificates containing a critical private extension. Since the extension is private, it will not be recognized by the PKIX `CertPathValidator` or `CertPathBuilder` implementation and a `CertPathValidatorException` will be thrown. In this case, a developer can implement a `PKIXCertPathChecker` that recognizes and processes the critical private extension.
- If the developer wants to record information about each certificate processed for debugging or display purposes.
- If the user wants to reject certificates with certain policy qualifiers.

The `setCertPathCheckers` method of the `PKIXParameters` class allows a user to pass a `List` of `PKIXCertPathChecker` objects to a PKIX `CertPathValidator` or `CertPathBuilder` implementation. Each of the `PKIXCertPathChecker` objects will be called in turn, for each certificate processed by the PKIX `CertPathValidator` or `CertPathBuilder` implementation.

Creating and using a PKIXCertPathChecker Object

The `PKIXCertPathChecker` class does not have a public constructor. This is intentional, since creating an instance of `PKIXCertPathChecker` is an implementation-specific issue. For example, the constructor for a `PKIXCertPathChecker` implementation that uses OCSP to check a certificate's revocation status may require the hostname and port of the OCSP server:

```
PKIXCertPathChecker checker = new OCSPChecker("ocsp.sun.com", 1321);
```

Once the checker has been instantiated, it can be added as a parameter using the `addCertPathChecker` method of the `PKIXParameters` class:

```
params.addCertPathChecker(checker);
```

Alternatively, a `List` of checkers can be added using the `setCertPathCheckers` method of the `PKIXParameters` class.

Implementing a PKIXCertPathChecker Object

The `PKIXCertPathChecker` class is abstract. It has four methods (`check`, `getSupportedExtensions`, `init`, and `isForwardCheckingSupported`) that all concrete subclasses must implement.

Implementing a `PKIXCertPathChecker` may be trivial or complex. A `PKIXCertPathChecker` implementation can be stateless or stateful. A stateless implementation does not maintain state between successive calls of the `check` method. For example, a `PKIXCertPathChecker` that checks that each certificate contains a particular policy qualifier is stateless. In contrast, a stateful implementation does maintain state between successive calls of the `check` method. The `check` method of a stateful implementation usually depends on the contents of prior

Copyright © 1993, 2026, Oracle and/or its affiliates.

certificates in the certification path. For example, a `PKIXCertPathChecker` that processes the NameConstraints extension is stateful.

Also, the order in which the certificates processed by a service provider implementation are presented (passed) to a `PKIXCertPathChecker` is very important, especially if the implementation is stateful. Depending on the algorithm used by the service provider, the certificates may be presented in *reverse* or *forward* order. A reverse ordering means that the certificates are ordered from the most trusted CA (if present) to the target subject, whereas a forward ordering means that the certificates are ordered from the target subject to the most trusted CA. The order must be made known to the `PKIXCertPathChecker` implementation, so that it knows how to process consecutive certificates.

Initializing a PKIXCertPathChecker Object

The `init` method initializes the internal state of the checker:

```
public abstract void init(boolean forward)
```

All stateful implementations should clear or initialize any internal state in the checker. This prevents a service provider implementation from calling a checker that is in an uninitialized state. It also allows stateful checkers to be reused in subsequent operations without reinstantiating them. The `forward` parameter indicates the order of the certificates presented to the `PKIXCertPathChecker`. If `forward` is `true`, the certificates are presented from target to trust anchor; if `false`, from trust anchor to target.

Forward Checking

The `isForwardCheckingSupported` method returns a `boolean` that indicates if the `PKIXCertPathChecker` supports forward checking:

```
public abstract boolean isForwardCheckingSupported()
```

All `PKIXCertPathChecker` implementations *must*support reverse checking. A `PKIXCertPathChecker` implementation *may*support forward checking.

Supporting forward checking improves the efficiency of `CertPathBuilders` that build forward, since it allows paths to be checked as they are built. However, some stateful `PKIXCertPathCheckers` may find it difficult or impossible to support forward checking.

Supported Extensions

The `getSupportedExtensions` method returns an immutable `Set` of OID `Strings` for the X.509 extensions that the `PKIXCertPathChecker` implementation supports (i.e., recognizes, is able to process):

```
public abstract Set<String> getSupportedExtensions()
```

The method should return `null` if no extensions are processed. All implementations should return the `Set` of OID `Strings` that the `check` method may process.

A `CertPathBuilder` can use this information to identify certificates with unrecognized critical extensions, even when performing a forward build with a `PKIXCertPathChecker` that does not support forward checking.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Executing the Check

The following method executes a check on the certificate:

```
public abstract void
        check(Certificate cert, Collection<String> unresolvedCritExts)
        throws CertPathValidatorException
```

The `unresolvedCritExts` parameter contains a collection of OIDs as `Strings`. These OIDs represent the set of critical extensions in the certificate that have not yet been resolved by the certification path validation algorithm. Concrete implementations of the `check` method should remove any critical extensions that it processes from the `unresolvedCritExts` parameter.

If the certificate does not pass the check(s), a `CertPathValidatorException` should be thrown.

Cloning a PKIXCertPathChecker

The `PKIXCertPathChecker` class implements the `Cloneable` interface. All stateful `PKIXCertPathChecker` implementations must override the `clone` method if necessary. The default implementation of the `clone` method calls the `Object.clone` method, which performs a simple clone by copying all fields of the original object to the new object. A stateless implementation should not override the `clone` method. However, all stateful implementations must ensure that the default `clone` method is correct, and override it if necessary. For example, a `PKIXCertPathChecker` that stores state in an array must override the `clone` method to make a copy of the array, rather than just a reference to the array.

The reason that `PKIXCertPathChecker` objects are `Cloneable` is to allow a PKIX `CertPathBuilder` implementation to efficiently backtrack and try another path when a potential certification path reaches a dead end or point of failure. In this case, the implementation is able to restore prior path validation states by restoring the cloned objects.

Example 9-3 Sample Code to Check for a Private Extension

This is an example of a stateless `PKIXCertPathChecker` implementation. It checks if a private extension exists in a certificate and processes it according to some rules.

```
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import java.util.Collection;
import java.util.Collections;
import java.util.Set;
import java.security.cert.PKIXCertPathChecker;
import java.security.cert.CertPathValidatorException;

public class MyChecker extends PKIXCertPathChecker {
    private static Set supportedExtensions =
        Collections.singleton("2.16.840.1.113730.1.1");

    /*
     * Initialize checker
     */
    public void init(boolean forward)
        throws CertPathValidatorException {
        // nothing to initialize
    }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    public Set getSupportedExtensions() {
        return supportedExtensions;
    }

    public boolean isForwardCheckingSupported() {
        return true;
    }

    /*
     * Check certificate for presence of Netscape's
     * private extension
     * with OID "2.16.840.1.113730.1.1"
     */
    public void check(Certificate cert,
                      Collection unresolvedCritExts)
        throws CertPathValidatorException
    {
        X509Certificate xcert = (X509Certificate) cert;
        byte[] ext =
            xcert.getExtensionValue("2.16.840.1.113730.1.1");
        if (ext == null)
            return;

        //
        // process private extension according to some
        // rules - if check fails, throw a
        // CertPathValidatorException ...
        // {insert code here}

        // remove extension from collection of unresolved
        // extensions (if it exists)
        if (unresolvedCritExts != null)
            unresolvedCritExts.remove("2.16.840.1.113730.1.1");
    }
}
```

How a PKIX Service Provider implementation should use a PKIXCertPathChecker

Each `PKIXCertPathChecker` object must be initialized by a service provider implementation before commencing the build or validation algorithm, for example:

```
List<PKIXCertPathChecker> checkers = params.getCertPathCheckers();
for (PKIXCertPathChecker checker : checkers) {
    checker.init(false);
}
```

For each certificate that it validates, the service provider implementation must call the `check` method of each `PKIXCertPathChecker` object in turn, passing it the certificate and any remaining unresolved critical extensions:

```
for (PKIXCertPathChecker checker : checkers) {
    checker.check(cert, unresolvedCritExts);
}
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

If any of the checks throw a CertPathValidatorException, a CertPathValidator implementation should terminate the validation procedure. However, a CertPathBuilder implementation may simply log the failure and continue to search for other potential paths. If all of the checks are successful, the service provider implementation should check that all critical extensions have been resolved and if not, consider the validation to have failed. For example:

```
if (unresolvedCritExts != null &&
    !unresolvedCritExts.isEmpty())
{
    // Note that a CertPathBuilder may have an enclosing
    // try block to catch the exception and continue on error
    throw new CertPathValidatorException
        ("Unrecognized Critical Extension");
}
```

As discussed in the previous section, a CertPathBuilder implementation may need to backtrack when a potential certification path reaches a dead end or point of failure. Backtracking in this context implies returning to the previous certificate in the path and checking for other potential paths. If the CertPathBuilder implementation is validating the path as it is building it, it will need to restore the previous state of each PKIXCertPathChecker. It can do this by making clones of the PKIXCertPathChecker objects *before* each certificate is processed, for example:

```
/* clone checkers */
List newList = new ArrayList(checkers);
ListIterator li = newList.listIterator();
while (li.hasNext()) {
    PKIXCertPathChecker checker = (PKIXCertPathChecker) li.next();
    li.set(checker.clone());
}
```

Using PKIXCertPathChecker in Certificate Path Validation

Using a PKIXCertPathChecker to customize certificate path validation is relatively straightforward.

Basic Certification Path Validation

First, consider code that validates a certificate path:

```
Set<TrustAnchor> trustAnchors = getTrustAnchors();
CertPath cp = getCertPath();

PKIXParameters pkixp = new PKIXParameters(trustAnchors);
pkixp.setRevocationEnabled(false);

CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
PKIXCertPathValidatorResult pcpvr =
    (PKIXCertPathValidatorResult)cpv.validate(cp, pkixp);
```

If the validation fails, the validate() method throws an exception.

The fundamental steps are as follows:

Copyright © 1993, 2026, Oracle and/or its affiliates.

1. Obtain the CA root certificates and the certification path to be validated.
2. Create a `PKIXParameters` with the trust anchors.
3. Use a `CertPathValidator` to validate the certificate path.

In this example, `getTrustAnchors()` and `getCertPath()` are the methods that obtain CA root certificates and the certification path.

The `getTrustAnchors()` method in the example must return a `Set` of `TrustAnchors` that represent the CA root certificates you wish to use for validation. Here is one simple implementation that loads a single CA root certificate from a file:

```
public Set<TrustAnchor> getTrustAnchors()
    throws IOException, CertificateException {

  CertificateFactory cf = CertificateFactory.getInstance("X.509");

  X509Certificate c;
  try (InputStream in = new FileInputStream("x509_ca-certificate.cer")) {
    c = (X509Certificate)cf.generateCertificate(in);
  }

  TrustAnchor anchor = new TrustAnchor(c, null);
  return Collections.singleton(anchor);
}
```

Similarly, here is a simple implementation of `getCertPath()` that loads a certificate path from a file:

```
public CertPath getCertPath() throws IOException, CertificateException {
  CertificateFactory cf = CertificateFactory.getInstance("X.509");

  CertPath cp;
  try (InputStream in = new FileInputStream("certpath.pkcs7")) {
    cp = cf.generateCertPath(in, "PKCS7");
  }
  return cp;
}
```

Note that PKCS#7 does not require a specific order for the certificates in the file, so this code only works for certification path validation when the certificates are ordered starting from the entity to be validated and progressing back toward the CA root. If the certificates are not in the right order, you need to do some additional processing. `CertificateFactory` has a `generateCertPath()` method that accepts a `Collection`, which is useful for this type of processing.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Adding in a `PKIXCertPathChecker`

To customize certification path validation, add a `PKIXCertPathChecker` as follows. In this example, `SimpleChecker` is a `PKIXCertPathChecker` subclass. The new lines are shown in **bold**.

```
Set<TrustAnchor> trustAnchors = getTrustAnchors();
CertPath cp = getCertPath();

PKIXParameters pkixp = new PKIXParameters(trustAnchors);
pkixp.setRevocationEnabled(false);

SimpleChecker sc = new SimpleChecker();
pkixp.addCertPathChecker(sc);

CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
PKIXCertPathValidatorResult pcpvr =
    (PKIXCertPathValidatorResult)cpv.validate(cp, pkixp);
```

`SimpleChecker` is a rudimentary subclass of `PKIXCertPathChecker`. Its `check()` method is called for every certificate in the certification path that is being validated. `SimpleChecker` uses an `AlgorithmConstraints` implementation to examine the signature algorithm and public key of each certificate.

```
import java.security.AlgorithmConstraints;
import java.security.CryptoPrimitive;
import java.security.Key;
import java.security.cert.*;
import java.util.*;

public class SimpleChecker extends PKIXCertPathChecker {
  private final static Set<CryptoPrimitive> SIGNATURE_PRIMITIVE_SET =
      EnumSet.of(CryptoPrimitive.SIGNATURE);

  public void init(boolean forward) throws CertPathValidatorException {}

  public boolean isForwardCheckingSupported() { return true; }

  public Set<String> getSupportedExtensions() { return null; }

  public void check(Certificate cert,
      Collection<String> unresolvedCritExts)
      throws CertPathValidatorException {
    X509Certificate c = (X509Certificate)cert;
    String sa = c.getSigAlgName();
    Key key = c.getPublicKey();

    AlgorithmConstraints constraints = new SimpleConstraints();

    if (constraints.permits(SIGNATURE_PRIMITIVE_SET, sa, null) == false)
      throw new CertPathValidatorException("Forbidden algorithm: " + sa);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    if (constraints.permits(SIGNATURE_PRIMITIVE_SET, key) == false)
      throw new CertPathValidatorException("Forbidden key: " + key);
  }
}
```

Finally, `SimpleConstraints` is an `AlgorithmConstraints` implementation that requires RSA 2048.

```
import java.security.AlgorithmConstraints;
import java.security.AlgorithmParameters;
import java.security.CryptoPrimitive;
import java.security.Key;
import java.security.interfaces.RSAKey;
import java.util.Set;

public class SimpleConstraints implements AlgorithmConstraints {
  public boolean permits(Set<CryptoPrimitive> primitives,
      String algorithm, AlgorithmParameters parameters) {
    return permits(primitives, algorithm, null, parameters);
  }

  public boolean permits(Set<CryptoPrimitive> primitives, Key key) {
    return permits(primitives, null, key, null);
  }

  public boolean permits(Set<CryptoPrimitive> primitives,
      String algorithm, Key key, AlgorithmParameters parameters) {
    if (algorithm == null) algorithm = key.getAlgorithm();

    if (algorithm.indexOf("RSA") == -1) return false;

    if (key != null) {
      RSAKey rsaKey = (RSAKey)key;
      int size = rsaKey.getModulus().bitLength();
      if (size < 2048) return false;
    }

    return true;
  }
}
```

Check Revocation Status of Certificates with PKIXRevocationChecker Class

An instance of `PKIXRevocationChecker` checks the revocation status of certificates with the Online Certificate Status Protocol (OCSP) or Certificate Revocation Lists (CRLs).

The `PKIXRevocationChecker` (introduced in JDK 8), which is a subclass of `PKIXCertPathChecker`, checks the revocation status of certificates with the PKIX algorithm.

An instance of `PKIXRevocationChecker` checks the revocation status of certificates with the Online Certificate Status Protocol (OCSP) or Certificate Revocation Lists (CRLs). OCSP is described in RFC 2560 and is a network protocol for determining the status of a certificate. A CRL is a time-stamped list identifying revoked certificates, and RFC 5280 describes an algorithm for determining the revocation status of certificates using CRLs.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Each PKIX `CertPathValidator` and `CertPathBuilder` instance provides a default revocation implementation that is enabled by default. If you want more control over the revocation settings used by that implementation, use the `PKIXRevocationChecker` class.

Follow these general steps to check the revocation status of a certificate path with the `PKIXRevocationChecker` class:

1. Obtain a `PKIXRevocationChecker` instance by calling the `getRevocationChecker` method of a PKIX `CertPathValidator` or `CertPathBuilder` instance.
2. Set additional parameters and options specific to certificate revocation with methods contained in the `PKIXRevocationChecker` class. These methods include `setOCSPResponder(URI)`, which sets the URI that identifies the location of the OCSP responder (although normally the URI is included in the certificate and does not have to be set) and `setOptions(Set<PKIXRevocationChecker.Option>)`, which sets revocation options. `PKIXRevocationChecker.Option` is an enumerated type used to specify the following options:
   - `ONLY_END_ENTITY`: Only check the revocation status of end-entity certificates.
   - `PREFER_CRLS`: By default, OCSP is the preferred mechanism for checking revocation status, with CRLs as the fallback mechanism. Switch this preference to CRLs with this option.
   - `SOFT_FAIL`: Ignore network failures.
3. After obtaining an instance of `PKIXRevocationChecker`, add it to a `PKIXParameters` or `PKIXBuilderParameters` object with the `addCertPathChecker` or `setCertPathCheckers` method.
4. Follow one of these steps depending on whether you are using a PKIX `CertPathValidator` or `CertPathBuilder` instance:
   - If you are using a PKIX `CertPathValidator` instance, call the `validate` method using as arguments the certificate path you want to validate and the `PKIXParameters` object that contains a revocation checker.
   - If you are using a PKIX `CertPathBuilder` instance, call the `build` method using as arguments the `PKIXBuilderParameters` object that contains a revocation checker.
5. Call the `validate` method of the PKIX `CertPathValidator` or `CertPathBuilder` instance using as arguments the certificate path you want to validate and the `PKIXParameters` or `PKIXBuilderParameters` object that contains a revocation checker.

The following excerpt checks the revocation status of certificates contained in a certificate path. The `CertPath` object `path` is the certificate path, and `params` is an object of type `PKIXParameters`:

```
    CertPathValidator cpv = CertPathValidator.getInstance("PKIX");
    PKIXRevocationChecker rc =
(PKIXRevocationChecker)cpv.getRevocationChecker();
    rc.setOptions(EnumSet.of(Option.SOFT_FAIL));
    params.addCertPathChecker(rc);
    params.setRevocationEnabled(false);
    CertPathValidatorResult res = cpv.validate(path, params);
```

In this excerpt, the `SOFT_FAIL` option causes the revocation checker to ignore any network failures (such as failing to establish a connection to the OCSP server) when it checks the revocation status.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Implementing a Service Provider

Experienced programmers can create their own provider packages supplying certification path service implementations.

This section assumes that you have read Java Cryptography Architecture (JCA) Reference Guide.

The following engine classes are defined in the Java Certification Path API:

- **`CertPathValidator`** - used to validate certification paths
- **`CertPathBuilder`** - used to build certification paths
- **`CertStore`** - used to retrieve certificates and CRLs from a repository

In addition, the pre-existing `CertificateFactory` engine class also supports the generation of certification paths.

The application interfaces supplied by an engine class are implemented in terms of a "Service Provider Interface" (SPI). The name of each SPI class is the same as that of the corresponding engine class, followed by "Spi". For example, the SPI class corresponding to the `CertPathValidator` engine class is the `CertPathValidatorSpi` class. Each SPI class is abstract. To supply the implementation of a particular type of service, for a specific algorithm or type, a provider must subclass the corresponding SPI class and provide implementations for all the abstract methods. For example, the `CertStore` class provides access to the functionality of retrieving certificates and CRLs from a repository. The actual implementation supplied in a `CertStoreSpi` subclass would be that for a specific type of certificate repository, such as LDAP.

Steps to Implement and Integrate a Provider

When implementing and integrating a provider for the certification path services, you must ensure that certain information is provided.

Developers should follow the Steps to Implement and Integrate a Provider. Here are some additional rules to follow for certain steps:

Step 3: Write your "Master Class", a subclass of Provider

In Step 3: Write Your Master Class, a Subclass of Provider these are the properties that must be defined for the certification path services, where the algorithm name is substituted for *algName*, and certstore type for *storeType*:

- `CertPathValidator.`*algName*
- `CertPathBuilder.`*algName*
- `CertStore.`*storeType*

See Java Security Standard Algorithm Names for the standard names that are defined for *algName* and *storeType*. The value of each property must be the fully qualified name of the class implementing the specified algorithm, or certstore type. That is, it must be the package name followed by the class name, where the two are separated by a period. For example, a

Copyright © 1993, 2026, Oracle and/or its affiliates.

provider sets the `CertPathValidator.PKIX` property to have the value `"sun.security.provider.certpath.PKIXCertPathValidator"` as follows:

```
put("CertPathValidator.PKIX",
"sun.security.provider.certpath.PKIXCertPathValidator")
```

In addition, service attributes can be defined for the certification path services. These attributes can be used as filters for selecting service providers. See Appendix A for the definition of some standard service attributes. For example, a provider may set the `ValidationAlgorithm` service attribute to the name of an RFC or specification that defines the PKIX validation algorithm:

```
put("CertPathValidator.PKIX ValidationAlgorithm", "RFC5280");
```

Step 11: Document your Provider and its Supported Services

In Step 12: Document Your Provider and Its Supported Services, certification path service providers should document the following information for each SPI:

Certificate Factories

A provider should document what types of certification paths (and the version numbers of the certificates in the path, if relevant) can be created by the factory. A provider should describe the ordering of the certificates in the certification path, as well as the contents.

A provider should document the list of encoding formats supported. This is not technically necessary, since the client can request them by calling the `getCertPathEncodings` method. However, the documentation should describe each encoding format in more detail and reference any standards when applicable.

Certification Path Validators

A provider should document any relevant information regarding the `CertPathValidator` implementation, including the types of certification paths that it validates. In particular, a PKIX `CertPathValidator` implementation should document the following information:

- The RFC or specification it is compliant with.
- The mechanism it uses to check that certificates have not been revoked.
- Any optional certificate or CRL extensions that it recognizes and how it processes them.

Certification Path Builders

A provider should document any relevant information regarding the `CertPathBuilder` implementation, including the types of certification paths that it creates and whether or not they are validated. In particular a PKIX `CertPathBuilder` implementation should document the following information:

- The RFC or specification it is compliant with.
- The mechanism it uses to check that certificates have not been revoked.
- Any optional certificate or CRL extensions that it recognizes and how it processes them.
- Details on the algorithm it uses for finding certification paths. Ex: depth-first, breadth-first, forward (i.e., from target to trust anchor(s)), reverse (i.e., from trust anchor(s) to target).

Copyright © 1993, 2026, Oracle and/or its affiliates.

- The algorithm it uses to select and sort potential certificates. For example, given two certificates that are potential candidates for the next certificate in the path, what criteria are used to select one before the other? What criteria are used to reject a certificate?
- If applicable, the algorithm it uses for backtracking or constructing another path (i.e., when potential paths do not meet constraints).
- The types of `CertStore` implementations that have been tested. The implementation should be designed to work with any `CertStore` type, but this information may still be useful.

All `CertPathBuilder` implementations should provide additional debugging support, in order to analyze and correct potential path building problems. Details on how to access this debugging information should be documented.

Certificate/CRL Stores

A provider should document what types of certificates and CRLs (and the version numbers, if relevant) are retrieved by the `CertStore`.

A provider should also document any relevant information regarding the `CertStore` implementation (such as protocols used or formats supported). For example, an LDAP `CertStore` implementation should describe which versions of LDAP are supported and which standard attributes are used for finding certificates and CRLs. It should also document if the implementation caches results, and for how long (i.e., under what conditions are they refreshed).

If the implementation returns the certificates and CRLs in a particular order, it should describe the sorting algorithm. An implementation should also document any additional or default initialization parameters. Finally, an implementation should document if and how it uses information in the `CertSelector` or `CRLSelector` objects to find certificates and CRLs.

Service Interdependencies

Common types of algorithm interdependencies in certification path service implementations.

The following are some common types of algorithm interdependencies in certification path service implementations:

- Certification Path Validation and Signature Algorithms

  A `CertPathValidator` implementation often requires use of a signature algorithm to verify each certificate's digital signature. The `setSigProvider` method of the `PKIXParameters` class allows a user to specify a specific `Signature` provider.

- Certification Path Builders and Certificate Factories

  A `CertPathBuilder` implementation will often utilize a `CertificateFactory` to generate a certification path from a list of certificates.

- CertStores and Certificate Factories

  A `CertStore` implementation will often utilize a `CertificateFactory` to generate certificates and CRLs from their encodings. For example, an LDAP `CertStore` implementation may use an X.509 `CertificateFactory` to generate X.509 certificates and CRLs from their ASN.1 encoded form.

Certification Path Parameter Specification Interfaces

The Certification Path API contains two interfaces representing *transparent* specifications of parameters, the `CertPathParameters` and `CertStoreParameters` interfaces.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Two implementations of the `CertPathParameters` interface are included, the `PKIXParameters` and `PKIXBuilderParameters` classes. If you are working with PKIX certification path validation and algorithm parameters, you can utilize these classes. If you need parameters for a different algorithm, you will need to supply your own `CertPathParameters` implementation for that algorithm.

Two implementations of the `CertStoreParameters` interface are included, the `LDAPCertStoreParameters` and the `CollectionCertStoreParameters` classes. These classes are to be used with LDAP and Collection `CertStore` implementations, respectively. If you need parameters for a different repository type, you will need to supply your own `CertStoreParameters` implementation for that type.

The `CertPathParameters` and `CertStoreParameters` interfaces each define a `clone` method that implementations should override. A typical implementation will perform a "deep" copy of the object, such that subsequent changes to the copy will not affect the original (and vice versa). However, this is not an absolute requirement for implementations of `CertStoreParameters`. A shallow copy implementation of `clone` is more appropriate for applications that need to hold a reference to a parameter contained in the `CertStoreParameters`. For example, since `CertStore.getInstance` makes a clone of the specified `CertStoreParameters`, a shallow copy `clone` allows an application to hold a reference to and later release the resources of a particular `CertStore` initialization parameter, rather than waiting for the garbage collection mechanism. This should be done with the utmost care, since the `CertStore` may still be in use by other threads.

Certification Path Result Specification Interfaces

The Certification Path API contains two interfaces representing *transparent* specifications of results, the `CertPathValidatorResult` and `CertPathBuilderResult` interfaces.

One implementation for each of the interfaces is included: the `PKIXCertPathValidatorResult` and `PKIXCertPathBuilderResult` classes. If you are implementing PKIX certification path service providers, you can utilize these classes. If you need certification path results for a different algorithm, you will need to supply your own `CertPathValidatorResult` or `CertPathBuilderResult` implementation for that algorithm.

A PKIX implementation of a `CertPathValidator` or a `CertPathBuilder` may find it useful to store additional information in the `PKIXCertPathValidatorResult` or `PKIXCertPathBuilderResult`, such as debugging traces. In these cases, the implementation should implement a subclass of the appropriate result class with methods to retrieve the relevant information. These classes must be shipped with the provider classes, for example, as part of the provider JAR file.

Certification Path Exception Classes

The Certification Path API contains a set of exception classes for handling errors. `CertPathValidatorException, CertPathBuilderException`, and `CertStoreException` are subclasses of `GeneralSecurityException`.

You may need to extend these classes in your service provider implementation.

For example, a `CertPathBuilder` implementation may provide additional information such as debugging traces when a `CertPathBuilderException` is thrown. The implementation may throw a subclass of `CertPathBuilderException` that holds this information. Likewise, a `CertStore` implementation can provide additional information when a failure occurs by throwing a subclass of `CertStoreException` . Also, you may want to implement a subclass of

Copyright © 1993, 2026, Oracle and/or its affiliates.

`CertPathValidatorException` to describe a particular failure mode of your `CertPathValidator` implementation.

In each case, the new exception classes must be shipped with the provider classes, for example, as part of the provider JAR file. Each provider should document the exception subclasses.

Appendix A: Standard Names

The Java Certification Path API requires and utilizes a set of standard names for certification path validation algorithms, encodings and certificate storage types.

The standard names previously found here in Appendix A and in the other security specifications (JCA/JSSE/etc.) have been combined in the Java Security Standard Algorithm Names. Specific provider information can be found in the JDK Providers.

Please note that a service provider may choose to define a new name for a proprietary or non-standard algorithm that is not mentioned in the Standard Names document. However, to prevent name collisions, it is recommended that the name be prefixed with the reverse Internet domain name of the provider's organization (for example: `com.sun.MyCertPathValidator`).

Appendix B: The CertPath Implementation in the SUN Provider

The "SUN" provider supports the following standard algorithms, types and encodings:

- `CertificateFactory`: **X.509** `CertPath` type with **PKCS7** and **PkiPath** encodings
- `CertPathValidator`: **PKIX** algorithm
- `CertPathBuilder`: **PKIX** algorithm
- `CertStore`: **Collection** `CertStore` type

The following discusses each of these service provider interface implementations:

CertificateFactory

The "SUN" provider for the `CertificateFactory` engine class supports generation of X.509 `CertPath` objects. The PKCS7 and PkiPath encodings are supported. The PKCS#7 implementation supports a subset of RFC 2315 (only the SignedData ContentInfo type is supported). The certificates in the `CertPath` are ordered in the forward direction (from target to trust anchor). Each certificate in the `CertPath` is of type `java.security.cert.X509Certificate` , and versions 1, 2 and 3 are supported.

CertPathValidator

The "SUN" provider supplies a PKIX implementation of the `CertPathValidator` engine class. The implementation validates `CertPaths` of type X.509 and implements the certification path validation algorithm defined in RFC 5280: *PKIX Certificate and CRL Profile*. This implementation sets the `ValidationAlgorithm` service attribute to "RFC5280".

Weak cryptographic algorithms can be disabled in the "SUN" provider using the `jdk.certpath.disabledAlgorithms` Security Property. See Appendix E: Disabling Cryptographic Algorithms for a description and examples of this property.

The PKIX Certificate and CRL Profile has many optional features. The "SUN" provider implements support for the policy mapping, authority information access and CRL distribution point certificate extensions, the issuing distribution point CRL extension, and the reason code and certificate issuer CRL entry extensions. It does not implement support for the freshest CRL

Copyright © 1993, 2026, Oracle and/or its affiliates.

or subject information access certificate extensions. It also does not include support for the freshest CRL and delta CRL Indicator CRL extensions and the invalidity date and hold instruction code CRL entry extensions.

The implementation supports a CRL revocation checking mechanism that conforms to section 6.3 of the PKIX Certificate and CRL Profile. OCSP (RFC 2560) is also currently supported as a built in revocation checking mechanism. See Appendix C: OCSP Support for more details on the implementation and configuration and how it works in conjunction with CRLs.

The implementation does not support the `nameConstraints` parameter of the `TrustAnchor` class and the `validate` method throws an `InvalidAlgorithmParameterException` if it is specified.

CertPathBuilder

The "SUN" provider supplies a PKIX implementation of the `CertPathBuilder` engine class. The implementation builds `CertPaths` of type X.509. Each `CertPath` is validated according to the PKIX algorithm defined in RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile . This implementation sets the `ValidationAlgorithm` service attribute to "RFC5280".

The implementation requires that the `targetConstraints` parameter of a `PKIXBuilderParameters` object is an instance of `X509CertSelector` and the subject criterion is set to a non-null value. Otherwise the `build` method throws an `InvalidAlgorithmParameterException`.

The implementation builds `CertPath` objects in a forward direction using a depth-first algorithm. It backtracks to previous states and tries alternate paths when a potential path is determined to be invalid or exceeds the `PKIXBuilderParameters` `maxPathLength` parameter.

Validation of the path is performed in the same manner as the `CertPathValidator` implementation. The implementation validates most of the path as it is being built, in order to eliminate invalid paths earlier in the process. Validation checks that cannot be executed on certificates ordered in a forward direction are delayed and executed on the path after it has been constructed (but before it is returned to the application).

As with `CertPathValidator`, the `jdk.certpath.disabledAlgorithms` Security Property can be used to exclude cryptographic algorithms that are not considered safe.

When two or more potential certificates are discovered that may lead to finding a path that meets the specified constraints, the implementation uses the following criteria in the order listed to prioritize the certificates:

1. The subject key identifier matches the authority key identifier of the current certificate.
2. The issuer DN matches the subject DN of a trust anchor.
3. The issuer DN is in the same namespace as a trust anchor, ordered by the most RDNs in common, unless the only RDN in common is geographical (for example, `C=US`). For example, if the trust anchor has a subject DN of `OU=D, OU=C, O=B, C=US`, then the implementation prioritizes the following certificates with these issuer DNs in the order listed:
    a. `OU=G, OU=C, O=B, C=US` – contains three RDNs in common
    b. `OU=H, O=B, C=US` – contains two RDNs in common
    c. `OU=H, O=D, C=US` – skipped and not prioritized because the only common ancestor, C=US is geographical
4. Any other certificate that doesn't meet the previous criteria.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Debugging support can be enabled by setting the `java.security.debug` property to `certpath`. For example:

```
java -Djava.security.debug=certpath BuildCertPath
```

This will print additional debugging information to standard error.

Collection CertStore

The SUN provider supports the Collection implementation of the `CertStore` engine class.

The Collection `CertStore` implementation can hold any objects that are an instance of `java.security.cert.Certificate` or `java.security.cert.CRL`.

The certificates and CRLs are not returned in any particular order and will not contain duplicates.

Support for the CRL Distribution Points Extension

Support for the CRL Distribution Points extension is available. It is disabled by default for compatibility and can be enabled by setting the system property `com.sun.security.enableCRLDP` to the value true.

If set to true, Oracle's PKIX implementation uses the information in a certificate's CRL Distribution Points extension (in addition to `CertStores` that are specified) to find the CRL, provided the distribution point is an X.500 distinguished name or a URI of type ldap, http, or ftp.

> **Note**
>
> Depending on your network and firewall setup, it may be necessary to also configure your networking proxy servers.

Support for the Authority Information Access (AIA) Extension

Support for the `caIssuers` access method of the Authority Information Access (AIA) extension is available. It is disabled by default for compatibility and can be enabled by setting the system property `com.sun.security.enableAIAcaIssuers` to the value `true`.

If set to true, Oracle's PKIX implementation of `CertPathBuilder` uses the information in a certificate's AIA extension (in addition to `CertStore` instances that are specified) to find the issuing CA certificate, provided it is a URI of type `ldap`, `http`, or `ftp`.

> **Note**
>
> Depending on your network and firewall setup, it may be necessary to also configure your networking proxy servers.

The system and security property `com.sun.security.allowedAIALocations` enables you to define one or more filtering rules to be applied to URIs obtained from the AIA extension on X.509 certificates. These filter rules are applied specifically to the CA issuers access method. Any CA issuers URIs in X.509 certificates are only followed when the `com.sun.security.enableAIAcaIssuers` system property is enabled and the filter allows the URI.

Copyright © 1993, 2026, Oracle and/or its affiliates.

To set the rules, you must set either the `com.sun.security.allowedAIALocations` security property or the system property by the same name. If the system property has a value, it will override the security property. By default the property is blank, which enacts a deny-all ruleset.

See the `java.security` file for more information about how to set this property.

Maximum Network Connection and Read Timeouts for CRL and OCSP Retrievals

The following system properties set timeout values in seconds or milliseconds. A number by itself or appended by `s` represents a value in seconds. For example, `10` and `10s` both represent 10 seconds. A number appended by `ms` represents a value in milliseconds. For example, `1000ms` represents 1000 milliseconds or 1 second. If a property has not been set, or if its value is negative, it is set to the default value of 15 seconds. A value of 0 means an infinite timeout.

Table 9-1 Maximum Network Connection and Read Timeouts for CRL and OCSP Retrievals

System Property	Description	Default Timeout Duration
`com.sun.security.cert.timeout`	Sets the maximum timeout to establish a TCP connection when specified in an X.509 certificate's `AuthorityInfoAccess` extension.	15 seconds
`com.sun.security.cert.readtimeout`	Sets the maximum timeout to read data once a TCP connection has been established when specified in an X.509 certificate's `AuthorityInfoAccess` extension.	15 seconds
`com.sun.security.crl.timeout`	Sets the maximum connection timeout for CRL retrievals.	15 seconds
`com.sun.security.crl.readtimeout`	Sets the maximum read timeout for CRL retrievals. The read timeout prevents a CRL download from taking a long time when a connection is established.	15 seconds
`com.sun.security.ocsp.timeout`	Sets the maximum timeout to establish a TCP connection.	15 seconds
`com.sun.security.ocsp.readtimeout`	Sets the maximum timeout to read data once a TCP connection has been established.	The value of `com.sun.security.ocsp.timeout`

Appendix C: OCSP Support

Client-side support for the On-Line Certificate Status Protocol (OCSP) as defined in RFC 2560 is supported.

OCSP checking is controlled by the following Security Properties:

Property Name	Description
`ocsp.enable`	This property's value is either true or false. If true, OCSP checking is enabled when doing certificate revocation checking; if false or not set, OCSP checking is disabled.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Property Name	Description
`ocsp.responderURL`	This property's value is a URL that identifies the location of the OCSP responder. Here is an example `ocsp.responderURL=http://ocsp.example.net:80` By default, the location of the OCSP responder is determined implicitly from the certificate being validated. The property is used when the Authority Information Access extension (defined in RFC 5280) is absent from the certificate or when it requires overriding.
`ocsp.responderCertSubjectName`	This property's value is the subject name of the OCSP responder's certificate. Here is an example `ocsp.responderCertSubjectName="CN=OCSP Responder, O=XYZ Corp"` By default, the certificate of the OCSP responder is that of the issuer of the certificate being validated. This property identifies the certificate of the OCSP responder when the default does not apply. Its value is a string distinguished name (defined in RFC 2253) which identifies a certificate in the set of certificates supplied during cert path validation. In cases where the subject name alone is not sufficient to uniquely identify the certificate, then both the `ocsp.responderCertIssuerName` and `ocsp.responderCertSerialNumber` properties must be used instead. When this property is set, then those two properties are ignored.
`ocsp.responderCertIssuerName`	This property's value is the issuer name of the OCSP responder's certificate . Here is an example `ocsp.responderCertIssuerName="CN=Enterprise CA, O=XYZ Corp"` By default, the certificate of the OCSP responder is that of the issuer of the certificate being validated. This property identifies the certificate of the OCSP responder when the default does not apply. Its value is a string distinguished name (defined in RFC 2253) which identifies a certificate in the set of certificates supplied during cert path validation. When this property is set then the `ocsp.responderCertSerialNumber` property must also be set. Note that this property is ignored when the `ocsp.responderCertSubjectName` property has been set.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Property Name	Description
`ocsp.responderCertSerialNumber`	This property's value is the serial number of the OCSP responder's certificate Here is an example `ocsp.responderCertSerialNumber=2A:FF:00` By default, the certificate of the OCSP responder is that of the issuer of the certificate being validated. This property identifies the certificate of the OCSP responder when the default does not apply. Its value is a string of hexadecimal digits (colon or space separators may be present) which identifies a certificate in the set of certificates supplied during cert path validation. When this property is set then the `ocsp.responderCertIssuerName` property must also be set. Note that this property is ignored when the `ocsp.responderCertSubjectName` property has been set.

These properties may be set either statically in the Java runtime's `<java_home>/conf/security/java.security` file, or dynamically using the `java.security.Security.setProperty()` method.

By default, OCSP checking is not enabled. It is enabled by setting the `ocsp.enable` property to `"true"`. Use of the remaining properties is optional. Note that enabling OCSP checking only has an effect if revocation checking has also been enabled. Revocation checking is enabled via the `PKIXParameters.setRevocationEnabled()` method.

OCSP checking works in conjunction with Certificate Revocation Lists (CRLs) during revocation checking. The following is a summary of the interaction of OCSP and CRLs. Failover to CRLs occurs only if an OCSP problem is encountered. Failover does not occur if the OCSP responder confirms either that the certificate has been revoked or that it has not been revoked.

PKIXParameters RevocationEnabled (default=true)	`ocsp.enable` (default=false)	Behavior
true	true	Revocation checking using OCSP, failover to using CRLs
true	false	Revocation checking using CRLs only
false	true	No revocation checking
false	false	No revocation checking

Enable OSCP Nonce Extension

The OSCP nonce extension cryptographically binds a request and a response to prevent replay attacks. The system property `jdk.security.certpath.ocspNonce` enables you to specify whether the `PKIXRevocationChecker` class includes the OCSP nonce extension:

Copyright © 1993, 2026, Oracle and/or its affiliates.

The OSCP nonce extension cryptographically binds a request and a response to prevent replay attacks. The system property `jdk.security.certpath.ocspNonce` enables you to specify whether the `PKIXRevocationChecker` class includes the OCSP nonce extension:

- If `jdk.security.certpath.OCSPNonce` is `false` and the application provides the OCSP nonce extension, then the `PKIXRevocationChecker` class uses application-provided nonce in the OCSP request.
- If `jdk.security.certpath.OCSPNonce` is `true` and the application does not provide the nonce extension, then the `PKIXRevocationChecker` class includes a 16-byte nonce with each OCSP request.
- If `jdk.security.certpath.OCSPNonce` is `true` and the application also provides the nonce extension, then the `PKIXRevocationChecker` class throws an exception.

By default, `jdk.security.certpath.OCSPNonce` is `false`.

Maximum Allowable Clock Skew

You might encounter connection failures during revocation checking because the network is slow or the system clock is off by some amount. Set the maximum allowable clock skew (the time difference between response time and local time), in seconds, used for revocation checks with the system property `com.sun.security.ocsp.clockskew`. If the property has not been set, or if its value is negative, it's set to the default value of 900 seconds (15 minutes).

Fallback Option for POST-Only OCSP Requests

HTTP-based OCSP requests can use either the GET or the POST method to submit their requests. By default, the OCSP client uses GET requests for small requests, which are those that are less than 255 bytes after encoding. It uses POST requests for everything else.

However, if your OCSP responders are encountering issues with GET requests, you can disable GET OCSP requests with the JDK system property - `Dcom.sun.security.ocsp.useget=false`. The default value of this property is `true`.

Appendix D: CertPath Implementation in JdkLDAP Provider

The JdkLDAP provider supports the LDAP implementation of the `CertStore` engine class.

LDAP CertStore

The LDAP `CertStore` implementation retrieves certificates and CRLs from an LDAP directory using the LDAP schema defined in RFC 2587.

The LDAPSchema service attribute is set to "RFC2587".

The implementation fetches certificates from different locations, depending on the values of the subject, issuer, and basicConstraints selection criteria specified in the `X509CertSelector`. It performs as many of the following operations as possible:

1. Subject non-null, basicConstraints <= -1
   Looks for certificates in the subject DN's "userCertificate" attribute.
2. Subject non-null, basicConstraints >= -1
   Looks for certificates in the forward element of the subject DN's "crossCertificatePair" attribute AND in the subject's "caCertificate" attribute.
3. Issuer non-null, basicConstraints >= -1

Copyright © 1993, 2026, Oracle and/or its affiliates.

Looks for certificates in the reverse element of the issuer DN's "crossCertificatePair" attribute AND in the issuer DN's "caCertificate" attribute.

In each case, certificates are checked using `X509CertSelector.match()` before adding them to the resulting collection.

If none of the conditions specified previously applies, then an exception is thrown to indicate that it was impossible to fetch certificates using the criteria supplied. Note that even if one or more of the conditions apply, the Collection returned may still be empty if there are no certificates in the directory.

The implementation fetches CRLs from the issuer DNs specified in the `setCertificateChecking`, `addIssuerName` or `setIssuerNames` methods of the `X509CRLSelector` class. If no issuer DNs have been specified using one of these methods, the implementation throws an exception indicating it was impossible to fetch CRLs using the criteria supplied. Otherwise, the CRLs are searched as follows:

1. The implementation first creates a list of issuer names. If a certificate was specified in the `setCertificateChecking` method, it uses the issuer of that certificate. Otherwise, it uses the issuer names specified using the `addIssuerName` or `setIssuerNames` methods.
2. Next, the implementation iterates through the list of issuer names. For each issuer name, it searches first in the issuer's `"authorityRevocationList"` attribute and then, if no matching CRL was found there, in the issuer's `"certificateRevocationList"` attribute. One exception is that if the issuer name was obtained from the certificate specified in the `setCertificateChecking` method, it only checks the issuer's `"authorityRevocationList"` attribute if the specified certificate is a CA certificate.
3. All CRLs are checked using `X509CRLSelector.match()` before adding them to the resulting collection.
4. If no CRLs satisfying the selection criteria can be found, an empty Collection is returned.

Caching

By default each LDAP CertStore instance caches lookups for a maximum of 30 seconds. The cache lifetime can be changed by setting the system property `sun.security.certpath.ldap.cache.lifetime` to a value in seconds. A value of `0` disables the cache completely. A value of `-1` means unlimited lifetime.

Appendix E: Disabling Cryptographic Algorithms

The `jdk.certpath.disabledAlgorithms` Security Property contains a list of cryptographic algorithms and key size constraints that are considered weak or broken. Certificates and other data (CRLs, OCSPResponses) containing any of these algorithms or key sizes will be blocked during certification path building and validation. This property is used by Oracle's PKIX implementation, other implementations might not examine and use it.

The exact syntax of the `jdk.certpath.disabledAlgorithms` Security Property is described in the `java.security` file. See Disabled and Restricted Cryptographic Algorithms for more information about this property.

Administrators or users can modify the value of the `jdk.certpath.disabledAlgorithms` property to address additional security requirements. However, removing any of the current algorithms or key sizes is not recommended.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> The algorithm restrictions specified by this Security Property do not apply to trust anchors or self-signed certificates.

Copyright © 1993, 2026, Oracle and/or its affiliates.

10

Java SASL API Programming and Deployment Guide

Simple Authentication and Security Layer, or SASL, is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data. It is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

SASL is used by protocols, such as the Lightweight Directory Access Protocol, version 3 (LDAP v3), and the Internet Message Access Protocol, version 4 (IMAP v4) to enable pluggable authentication. Instead of hardwiring an authentication method into the protocol, LDAP v3 and IMAP v4 use SASL to perform authentication, thus enabling authentication via various SASL mechanisms.

There are a number of standard SASL mechanisms defined by the Internet community for various levels of security and deployment scenarios. These range from no security (for example, anonymous authentication) to high security (for example, Kerberos authentication) and levels in between.

The Java SASL API

The Java SASL API defines classes and interfaces for applications that use SASL mechanisms. It is defined to be mechanism-neutral: the application that uses the API need not be hardwired into using any particular SASL mechanism. The API supports both client and server applications. It allows applications to select the mechanism to use based on desired security features, such as whether they are susceptible to passive dictionary attacks or whether they accept anonymous authentication.

The Java SASL API also allows developers to use their own, custom SASL mechanisms. SASL mechanisms are installed by using the Java Cryptography Architecture (JCA); see Java Cryptography Architecture (JCA) Reference Guide.

When to Use SASL

SASL provides a pluggable authentication and security layer for network applications. There are other features in Java SE that provide similar functionality, including Java Secure Socket Extension (JSSE) (see Java Secure Socket Extension (JSSE) Reference Guide) and the Java Generic Security Service. JSSE provides a framework and an implementation for a Java language version of the SSL, TLS, and DTLS protocols. Java GSS is the Java language bindings for the Generic Security Service Application Programming Interface (GSS-API). The only mechanism currently supported underneath this API on Java SE is Kerberos v5.

With the exception of defining and building protocols from scratch, protocol definition is often the biggest factor that goes into determining which API to use. When compared with JSSE and Java GSS, SASL is relatively lightweight and is popular among some protocols. It also has the advantage that several popular, lightweight (in terms of infrastructure support) SASL mechanisms have been defined. Primary JSSE and Java GSS mechanisms, on the other hand, have relatively heavyweight mechanisms that require more elaborate infrastructures (Public Key Infrastructure and Kerberos, respectively).

Copyright © 1993, 2026, Oracle and/or its affiliates.

SASL, JSSE, and Java GSS are often used together. For example, a common pattern is for an application to use JSSE for establishing a secure channel, and to use SASL for client, username/password-based authentication. There are also SASL mechanisms layered on top of GSS-API mechanisms; one popular example is a SASL GSS-API/Kerberos v5 mechanism that is used with LDAP.

With the exception of defining and building protocols from scratch, protocol definition is often the biggest factor in determining which API to use. For example, LDAP and IMAP are defined to use SASL, so software related to these protocols should use the Java SASL API. When building Kerberos applications and services, the API to use is Java GSS. When building applications and services that use SSL/TLS as their protocol, the API to use is JSSE.

Java SASL API Overview

SASL is a challenge-response protocol. The server issues a challenge to the client, and the client sends a response based on the challenge. This exchange continues until the server is satisfied and issues no further challenge. These challenges and responses are binary tokens of arbitrary length. The encapsulating protocol (such as LDAP or IMAP) specifies how these tokens are encoded and exchanged. For example, LDAP specifies how SASL tokens are encapsulated within LDAP bind requests and responses.

The Java SASL API is modeled according to this style of interaction and usage. It has interfaces, `SaslClient` and `SaslServer`, that represent client-side and server-side mechanisms, respectively. The application interacts with the mechanisms via byte arrays that represent the challenges and responses. The server-side mechanism iterates, issuing challenges and processing responses, until it is satisfied, while the client-side mechanism iterates, evaluating challenges and issuing responses, until the server is satisfied. The application that is using the mechanism drives each iteration. That is, it extracts the challenge or response from a protocol packet and supplies it to the mechanism, and then puts the response or challenge returned by the mechanism into a protocol packet and sends it to the peer.

Creating the Mechanisms

The client and server code that use the SASL mechanisms are not hardwired to use specific mechanism(s). In many protocols that use SASL, the server advertises (either statically or dynamically) a list of SASL mechanisms that it supports. The client then selects one of these based on its security requirements.

The `Sasl` class is used for creating instances of `SaslClient` and `SaslServer`. Here is an example of how an application creates a SASL client mechanism using a list of possible SASL mechanisms.

```
String[] mechanisms = new String[]{"DIGEST-MD5", "PLAIN"};
SaslClient sc = Sasl.createSaslClient(
    mechanisms, authzid, protocol, serverName, props, callbackHandler);
```

Based on the availability of the mechanisms supported by the platform and other configuration information provided via the parameters, the Java SASL framework selects one of the listed mechanisms and return an instance of `SaslClient`.

The name of the selected mechanism is usually transmitted to the server via the application protocol. Upon receiving the mechanism name, the server creates a corresponding

Copyright © 1993, 2026, Oracle and/or its affiliates.

`SaslServer` object to process client-sent responses. Here is an example of how the server would create an instance of `SaslServer`.

```
SaslServer ss = Sasl.createSaslServer(
    mechanism, protocol, myName, props, callbackHandler);
```

Passing Input to the Mechanisms

Because the Java SASL API is a general framework, it must be able to accommodate many different types of mechanisms. Each mechanism needs to be initialized with input and may need input to make progress. The API provides three means by which an application gives input to a mechanism:

1. **Common input parameters**: The application uses predefined parameters to supply information that are defined by the SASL specification and commonly required by mechanisms. For`SaslClient` mechanisms, the input parameters are authorization id, protocol id, and server name. For`SaslServer` mechanisms, the common input parameters are protocol id and (its own fully qualified) server name.
2. **Properties parameter**: The application uses the properties parameter, a mapping of property names to (possibly non-string) property values, to supply configuration information. The Java SASL API defines some standard properties, such as `Sasl.QOP` (quality-of-protection), `Sasl.STRENGTH` (cipher strength), and `Sasl.MAX_BUFFER` (maximum buffer size). The parameter can also be used to pass in non-standard properties that are specific to particular mechanisms.
3. **Callbacks**: The application uses the `CallbackHandler` parameter to supply input that cannot be predetermined or might not be common across mechanisms. When a mechanism requires input data, it uses the callback handler supplied by the application to collect the data, possibly from the end-user of the application. For example, a mechanism might require the end-user of the application to supply a name and password.

   Mechanisms can use the callbacks defined in the `javax.security.auth.callback` package; these are generic callbacks useful for building applications that perform authentication. Mechanisms might also need SASL-specific callbacks, such as those for collecting realm and authorization information, or even (non-standardized) mechanism-specific callbacks. The application should be able to accommodate a variety of mechanisms. Consequently, its callback handler must be able to service all of the callbacks that the mechanisms might request. This is not possible in general for arbitrary mechanisms, but is usually feasible due to the limited number of mechanisms that are typically deployed and used.

Using the Mechanisms

Once the application has created a mechanism, it uses the mechanism to obtain SASL tokens to exchange with the peer. The client typically indicates to the server via the application protocol which mechanism to use. Some protocols allow the client to accompany the request with an optional initial response for mechanisms that have an initial response. This feature can be used to lower the number of message exchanges required for authentication. Here is an example of how a client might use `SaslClient` for authentication.

```
// Get optional initial response
byte[] response =
    (sc.hasInitialResponse() ? sc.evaluateChallenge(new byte[]) : null);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    String mechanism = sc.getMechanismName();

    // Send selected mechanism name and optional initial response to server
    send(mechanism, response);

    // Read response
    msg = receive();
    while (!sc.isComplete() && (msg.status == CONTINUE || msg.status ==
SUCCESS)) {
        // Evaluate server challenge
        response = sc.evaluateChallenge(msg.contents);

        if (msg.status == SUCCESS) {
            // done; server doesn't expect any more SASL data
             if (response != null) {
                throw new IOException(
                    "Protocol error: attempting to send response after
completion");
            }
            break;
        } else {
            send(mechanism, response);
            msg = receive();
        }
    }
```

The client application iterates through each step of the authentication by using the mechanism (`sc`) to evaluate the challenge gotten from the server and to get a response to send back to the server. It continues this cycle until either the mechanism or application-level protocol indicates that the authentication has completed, or if the mechanism cannot evaluate a challenge. If the mechanism cannot evaluate the challenge, it throws an exception to indicate the error and terminates the authentication. Disagreement between the mechanism and protocol about the completion state must be treated as an error because it might indicate a compromise of the authentication exchange.

Here is an example of how a server might use `SaslServer`.

```
    // Read request that contains mechanism name and optional initial response
    msg.receive();

    // Obtain a SaslServer to perform authentication
    SaslServer ss = Sasl.createSaslServer(msg.mechanism,
        protocol, myName, props, callbackHandler);

    // Perform authentication steps until done
    while (!ss.isComplete()) {
        try {
            // Process response
            byte[] challenge = sc.evaluateResponse(msg.contents);

            if (ss.isComplete()) {
                send(mechanism, challenge, SUCCESS);
            } else {
                send(mechanism, challenge, CONTINUE);
                msg.receive();
            }
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            } catch (SaslException e) {
                send(ERROR);
                sc.dispose();
                break;
            }
        }
```

The server application iterates through each step of the authentication by giving the client's response to the mechanism (`ss`) to process. If the response is incorrect, the mechanism indicates the error by throwing a `SaslException` so that the server can report the error and terminate the authentication. If the response is correct, the mechanism returns challenge data to be sent to the client and indicates whether the authentication is complete. Note that challenge data can accompany a "success" indication. This might be used, for example, to tell the client to finalize some negotiated state.

Using the Negotiated Security Layer

Some SASL mechanisms support only authentication while others support use of a negotiated security layer after authentication. The security layer feature is often not used when the application uses some other means, such as SSL/TLS, to communicate securely with the peer.

When a security layer has been negotiated, all subsequent communication with the peer must take place using the security layer. To determine whether a security layer has been negotiated, get the negotiated `Sasl.QOP` from the mechanism. Here is an example of how to determine whether a security layer has been negotiated.

```
String qop = (String) sc.getNegotiatedProperty(Sasl.QOP);
boolean hasSecurityLayer = (qop != null &&
    (qop.equals("auth-int") || qop.equals("auth-conf")));
```

A security layer has been negotiated if the `Sasl.QOP` property indicates that either integrity and/or confidentiality has been negotiated.

To communicate with the peer using the negotiated layer, the application first uses the `wrap` method to encode the data to be sent to the peer to produce a "wrapped" buffer. It then transfers a length field representing the number of octets in the wrapped buffer followed by the contents of the wrapped buffer to the peer. The peer receiving the stream of octets passes the buffer (without the length field) to `unwrap` to obtain the decoded bytes sent by the peer. Details of this protocol are described in RFC 2222. Example 10-1 illustrates how a client application sends and receives application data using a security layer.

Example 10-1 Sample Code for SASL Client Send and Receive Data

```
// Send outgoing application data to peer
byte[] outgoing = ...;
byte[] netOut = sc.wrap(outgoing, 0, outgoing.length);

send(netOut.length, netOut);   // send to peer

// Receive incoming application data from peer
byte[] netIn = receive();      // read length and ensuing bytes from peer

byte[] incoming = sc.unwrap(netIn, 0, netIn.length);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

How SASL Mechanisms are Installed and Selected

SASL mechanism implementations are provided by SASL security providers. Each provider may support one or more SASL mechanisms and is registered with the JCA.

By default, the SunSASL provider is automatically registered as a JCA provider. To remove it or reorder its priority as a JCA provider, change the line

```
security.provider.7=SunSASL
```

in the Java security properties file (*java-home*`/conf/security/java.security`).

To add or remove a SASL provider, you add or remove the corresponding line in the security properties file. For example, if you want to add a SASL provider and have its mechanisms be chosen over the same ones implemented by the SunSASL provider, then you would add a line to the security properties file with a lower number.

```
security.provider.7=com.example.MyProvider
security.provider.8=SunSASL
```

Alternatively, you can programmatically add your own provider using the `java.security.Security` class. For example, the following sample code registers the `com.example.MyProvider` to the list of available SASL security providers.

```
Security.addProvider(new com.example.MyProvider());
```

See Step 8: Prepare for Testing in Steps to Implement and Integrate a Provider for more information about adding providers to the security properties file and programmatically adding your own providers.

When an application requests a SASL mechanism by supplying one or more mechanism names, the SASL framework looks for registered SASL providers that support that mechanism by going through, in order, the list of registered providers. The providers must then determine whether the requested mechanism matches the selection policy properties in the `Sasl` and if so, return an implementation for the mechanism.

The selection policy properties specify the security aspects of a mechanism, such as its susceptibility to certain attacks. These are characteristics of the mechanism (definition), rather than its implementation so all providers should come to the same conclusion about a particular mechanism. For example, the PLAIN mechanism is susceptible to plaintext attacks regardless of how it is implemented. If no selection policy properties are supplied, there are no restrictions on the selected mechanism. Using these properties, an application can ensure that it does not use unsuitable mechanisms that might be deployed in the execution environment. For example, an application might use the following sample code if it does not want to allow the use of mechanisms susceptible to plaintext attacks.

```
Map<String, String> props = new HashMap<>();
props.put(Sasl.POLICY_NOPLAINTEXT, "true");
SaslClient sc = Sasl.createSaslClient(
    mechanisms, authzid, protocol, serverName, props, callbackHandler);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The SunSASL Provider

The SunSASL provider supports the following client and server mechanisms:

- Client Mechanisms
  - PLAIN (RFC 2595). This mechanism supports cleartext user name/password authentication.
  - CRAM-MD5 (RFC 2195). This mechanism supports a hashed user name/password authentication scheme.
  - DIGEST-MD5 (RFC 2831). This mechanism defines how HTTP Digest Authentication can be used as a SASL mechanism.
  - EXTERNAL (RFC 2222). This mechanism obtains authentication information from an external channel (such as TLS or IPsec).
  - NTLM. This mechanism supports NTLM authentication.
- Server Mechanisms
  - CRAM-MD5
  - DIGEST-MD5
  - NTLM

The SunSASL Provider Client Mechanisms

The SunSASL provider supports several SASL client mechanisms used in popular protocols such as LDAP, IMAP, and SMTP.

The following table summarizes the client mechanisms and their required input.

Table 10-1 SunSASL Provider Client Mechanisms

Client Mechanism Name	Parameters/Input	Callbacks	Configuration Properties	Selection Policy
CRAM-MD5	authorization id (as default user name)	`PasswordCallback` `NameCallback`	None	`Sasl.POLICY_NOANONYMOUS` `Sasl.POLICY_NOPLAINTEXT`

DIGEST-MD5	authorization id
	protocol id
	server name
		`NameCallback`
		`PasswordCallback`
		`RealmCallback`
		`RealmChoiceCallback`
			`Sasl.QOP`
			`Sasl.STRENGTH`
			`Sasl.MAX_BUFFER`
			`Sasl.SERVER_AUTH`
			`javax.security.sasl.sendmaxbuffer`
			`com.sun.security.sasl.digest.cipher`
				`Sasl.POLICY_NOANONYMOUS`
				`Sasl.POLICY_NOPLAINTEXT`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 10-1 (Cont.) SunSASL Provider Client Mechanisms

Client Mechanism Name	Parameters/Input	Callbacks	Configuration Properties	Selection Policy
EXTERNAL	authorization id external channel	None	None	Sasl.POLICY_NOPLAINTEXT Sasl.POLICY_NOACTIVE Sasl.POLICY_NODICTIONARY
NTLM	authzId (as default user name) serverName (as default domain)	RealmCallback NameCallback PasswordCallback	Sasl.QOP com.sun.security.sasl.ntlm.version com.sun.security.sasl.ntlm.random com.sun.security.sasl.ntlm.hostname	Sasl.POLICY_NOANONYMOUS Sasl.POLICY_NOPLAINTEXT
PLAIN	authorization id	NameCallback PasswordCallback	None	Sasl.POLICY_NOANONYMOUS

An application that uses these mechanisms from the SunSASL provider must supply the required parameters, callbacks and properties. The properties have reasonable defaults and only need to be set if the application wants to override the defaults. Most of the parameters, callbacks, and properties are described in the API documentation. The following sections describe mechanism-specific behaviors and parameters not already covered by the API documentation.

Cram-MD5

The Cram-MD5 client mechanism uses the authorization id parameter, if supplied, as the default user name in the `NameCallback` to solicit the application/end-user for the authentication id. The authorization id is otherwise not used by the Cram-MD5 mechanism; only the authentication id is exchanged with the server.

Digest-MD5

The Digest-MD5 mechanism is used for digest authentication and optional establishment of a security layer. It specifies the following ciphers for use with the security layer: Triple DES, DES and RC4 (128, 56, and 40 bits). The Digest-MD5 mechanism can support only ciphers that are available on the platform. For example, if the platform does not support the RC4 ciphers, then the Digest-MD5 mechanism will not use those ciphers.

The `Sasl.STRENGTH` property supports `high`, `medium`, and `low` settings; its default is `high,medium,low`. The ciphers are mapped to the strength settings as follows:

Table 10-2 Cipher Strength

Strength	Cipher	Cipher Id
high	Triple DES RC4 128 bits	3des rc4

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 10-2 (Cont.) Cipher Strength

Strength	Cipher	Cipher Id
medium	DES RC4 56 bits	des rc4-56
low	RC4 40 bits	rc4-40

When there is more than one choice for a particular strength, the cipher selected depends on the availability of the ciphers in the underlying platform. To explicitly name the cipher to use, set the `com.sun.security.sasl.digest.cipher` property to the corresponding cipher id. Note that this property setting must be compatible with `Sasl.STRENGTH` and the ciphers available in the underlying platform. For example, `Sasl.STRENGTH` being set to `low` and `com.sun.security.sasl.digest.cipher` being set to `3des` are incompatible. The `com.sun.security.sasl.digest.cipher` property has no default.

The `javax.security.sasl.sendmaxbuffer`property specifies (the string representation of) the maximum send buffer size in bytes. The default is 65536. The actual maximum number of bytes will be the minimum of this number and the peer's maximum receive buffer size.

NTLM

> **Note**
>
> This section applies both to the NTLM client mechanism and the NTLM server mechanism.

NT LAN Manager (NTLM) is an security protocol from Microsoft used to access their various services such as IIS Web Server and Exchange Mail Server. As a SASL mechanism, it can be used to access Microsoft Exchange Server. It is also useful for HTTP authentication with the NTLM scheme.

The NTLM mechanism is used for NTLM authentication. It does not provide a security layer. This means that you can only set the `javax.security.sasl.qop` environment property to `auth`.

If the LMCompatibilityLevel registry value is set to a high value on the server, certain low value requests are not supported. However, there's no protocol for the server to inform the client to use a higher version, so the user must manually choose the correct version on the client side.

Set the system property `ntlm.debug` to any value to turn on debugging

Provide the following information either at mechanism creation or through callbacks:

Table 10-3 NTLM Required Information

Information	Type	Required or Optional	Description
Name	`String`	Required	Provided through `NameCallback` with the `authzid` input argument as the default value

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 10-3 (Cont.) NTLM Required Information

Information	Type	Required or Optional	Description
Password	`char[]`	Required	Provided through `PasswordCallback` If the password contains non-ASCII characters, the original LM version might fail. In this case, do not choose `LM` as the version.
Domain	`String`	Optional	Provided through `RealmCallback` with the `serverName` input argument as the default value. The domain provided on the client side is used to create the Type 1 message. The negotiated property `com.sun.security.sasl.ntlm.domain` is determined by the server's Type 2 message.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 10-3 (Cont.) NTLM Required Information

Information	Type	Required or Optional	Description
NTLM version	`String`	Optional	Specifies a specific version to use. Provided through the `com.sun.security.sasl.ntlm.version` property. It can have one of the following values: • `LM/NTLM`: Original NTLM v1 • `LM`: Original NTLM v1, LM only • `NTLM`: Original NTLM v1, NTLM only • `NTLM2`: NTLM v1 with Client Challenge • `LMv2/NTLMv2`: NTLM v2 • `LMv2`: NTLM v2, LM only • `NTLMv2`: NTLM v2, NTLM only If not provided, then the system property `ntlm.version` is used. If still not provided, then the value `LMv2/NTLMv2` is used, and on the server side, all values are accepted. Note: these types are only different on the client side. On the server side, because authentication succeeds if only one of LM (or LMv2) or NTLM (or NTLMv2) is verified, the first three types are effectively the same; this is also true for the last three types.
Host name	`String`	Optional	Provided through the `com.sun.security.sasl.ntlm.hostname` property, which will be sent to the server. If not provided, then the system will automatically derive a host name. This property is only used on the client side.
Random source	`java.util.Random`	Optional	Used as random source to derive nonce bytes. Provided through the `com.sun.security.sasl.ntlm.random` property. If not provided, then an internal `java.util.Random` object is used.

Copyright © 1993, 2026, Oracle and/or its affiliates.

After authentication, the client will receive a negotiated property named `com.sun.security.sasl.html.domain`, which is provided by the server, and the server will receive a negotiated property named `com.sun.security.sasl.ntlm.hostname`, which is he host name the client used to access this server.

The SunSASL Provider Server Mechanisms

The SunSASL provider supports several SASL server mechanisms used in popular protocols such as LDAP, IMAP, and SMTP.

The following table summarizes the server mechanisms and the required input:

Table 10-4 Server Mechanisms

Server Mechanism Name	Parameters/Input	Callbacks	Configuration Properties	Selection Policy

CRAM-MD5	server name	`AuthorizeCallback`
		`NameCallback`
		`PasswordCallback`
			None	`Sasl.POLICY_NOANONYMOUS`
				`Sasl.POLICY_NOPLAINTEXT`

DIGEST-MD5	protocol id
	server name
		`AuthorizeCallback`
		`NameCallback`
		`PasswordCallback`
		`RealmCallback`
			`Sasl.QOP`
			`Sasl.STRENGTH`
			`Sasl.MAX_BUFFER`
			`javax.security.sasl.sendmaxbuffer`
			`com.sun.security.sasl.digest.realm`
			`com.sun.security.sasl.digest.utf8`
				`Sasl.POLICY_NOANONYMOUS`
				`Sasl.POLICY_NOPLAINTEXT`

NTLM	serverName (as domain, can be overridden by properties)	`RealmCallback`, providing request user's domain
		`NameCallback`, providing request user's name
		`PasswordCallback`
			`Sasl.QOP`
			`com.sun.security.sasl.ntlm.random`
			`com.sun.security.sasl.ntlm.version`
			`com.sun.security.sasl.ntlm.domain`
				`Sasl.POLICY_NOANONYMOUS`
				`Sasl.POLICY_NOPLAINTEXT`

An application that uses these mechanisms from the SunSASL provider must supply the required parameters, callbacks and properties. The properties have reasonable defaults and only need to be set if the application wants to override the defaults.

All users of server mechanisms must have a callback handler that deals with the `AuthorizeCallback`. This is used by the mechanisms to determine whether the authenticated user is allowed to act on behalf of the requested authorization id, and also to obtain the canonicalized name of the authorized user (if canonicalization is applicable).

Most of the parameters, callbacks, and properties are described in the API documentation. The following sections describe mechanism-specific behaviors and parameters not already covered by the API documentation.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Cram-MD5

The Cram-MD5 server mechanism uses the `NameCallback` and `PasswordCallback` to obtain the password required to verify the SASL client's response. The callback handler should use the `NameCallback.getDefaultName()` as the key to fetch the password.

Digest-MD5

The Digest-MD5 server mechanism uses the `RealmCallback`, `NameCallback`, and `PasswordCallback` to obtain the password required to verify the SASL client's response. The callback handler should use `RealmCallback.getDefaultText()` and `NameCallback.getDefaultName()` as keys to fetch the password.

The `javax.security.sasl.sendmaxbuffer` property specifies (the string representation of) the maximum send buffer size in bytes. The default is 65536. The actual maximum number of bytes will be the minimum of this number and the peer's maximum receive buffer size.

The `com.sun.security.sasl.digest.realm` property is used to specify a list of space-separated realm names that the server supports. The list is sent to the client as part of the challenge. If this property has not been set, the default realm is the server's name (supplied as a parameter).

The `com.sun.security.sasl.digest.utf8` property is used to specify the character encoding to use. The value `true` means to use UTF-8 encoding; the value `false` means to use ISO Latin 1 (ISO-8859-1). The default value is `true`.

The JdkSASL Provider

The JdkSASL provider supports the following client and server mechanisms:

- Client Mechanisms
  - GSSAPI (RFC 2222). This mechanism uses the GSSAPI for obtaining authentication information. It supports Kerberos v5 authentication.
- Server Mechanisms
  - GSSAPI (Kerberos v5)

The JdkSASL Provider Client Mechanism

The JdkSASL provider supports the GSSAPI client mechanism used in popular protocols such as LDAP, IMAP, and SMTP.

The following table summarizes the GSSAPI client mechanism and its required input.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 10-5 JdkSASL Provider Client Mechanism

Client Mechanism Name	Parameters/Input	Callbacks	Configuration Properties	Selection Policy
GSSAPI	JAAS Subject authorization id protocol id server name	None	Sasl.QOP Sasl.MAX_BUFFER Sasl.SERVER_AUTH javax.security.sasl.sendmaxbuffer com.sun.security.jgss.inquiretype.*type_name*	Sasl.POLICY_NOACTIVE Sasl.POLICY_NOANONYMOUS Sasl.POLICY_NOPLAINTEXT

An application that uses the GSSAPI mechanism from the JdkSASL provider must supply the required parameters, callbacks and properties. The properties have reasonable defaults and only need to be set if the application wants to override the defaults. Most of the parameters, callbacks, and properties are described in the API documentation. The following section describes further GSSAPI behaviors and parameters not already covered by the API documentation.

GSSAPI

> **Note**
>
> The GSSAPI server mechanism has the same requirements as the GSSAPI client mechanism in terms of Kerberos credentials and the `javax.security.sasl.sendmaxbuffer` property.

The GSSAPI mechanism is used for Kerberos v5 authentication and optional establishment of a security layer. The mechanism expects the calling thread's `Subject` to contain the client's Kerberos credentials or that the credentials could be obtained by implicitly logging in to Kerberos. To obtain the client's Kerberos credentials, use the Java Authentication and Authorization Service (JAAS) to log in using the Kerberos login module. See Introduction to JAAS and Java GSS-API Tutorials for details and examples. After using JAAS authentication to obtain the Kerberos credentials, you put the code that uses the SASL GSSAPI mechanism within `Subject.callAs(Subject, Callable<T>)`.

```
LoginContext lc = new LoginContext("JaasSample", new TextCallbackHandler());
lc.login();
lc.getSubject().callAs(new SaslAction());

class SaslAction implements java.util.concurrent.Callable<Void> {
    public Void call() {
        // ...
        String[] mechanisms = new String[]{"GSSAPI"};
        SaslClient sc = Sasl.createSaslClient(
            mechanisms, authzid, protocol, serverName, props,
callbackHandler);
        // ...
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
    }
}
```

To obtain Kerberos credentials without doing explicit JAAS programming, see Use of Java GSS-API for Secure Message Exchanges Without JAAS Programming. When using this approach, there is no need to wrap the code within `Subject.callAs(Subject, Callable<T>)`.

The `javax.security.sasl.sendmaxbuffer` property specifies (the string representation of) the maximum send buffer size in bytes. The default is 65536. The actual maximum number of bytes will be the minimum of this number and the peer's maximum receive buffer size.

The `com.sun.security.jgss.inquiretype.`*`type_name`* negotiated property contains the value returned by the `ExtendedGSSContext.inquireSecContext(InquireType)` method, where *`type_name`* is the string form of the `InquireType` enum parameter in lower case.

The JdkSASL Provider Server Mechanism

The JdkSASL provider supports the GSSAPI mechanism used in popular protocols such as LDAP, IMAP, and SMTP.

The following table summarizes the GSSAPI server mechanism and the required input:

Table 10-6 Server mechanism

Server Mechanism Name	Parameters/Input	Callbacks	Configuration Properties	Selection Policy

GSSAPI	`Subject`
	protocol id
	server name
		`AuthorizeCallback`	`Sasl.QOP`
			`Sasl.MAX_BUFFER`
			`javax.security.sasl.sendmaxbuffer`
				`Sasl.POLICY_NOACTIVE`
				`Sasl.POLICY_NOANONYMOUS`
				`Sasl.POLICY_NOPLAINTEXT`

An application that uses the GSSAPI mechanism from the JdkSASL provider must supply the required parameters, callbacks and properties. The properties have reasonable defaults and only need to be set if the application wants to override the defaults.

All users of server mechanism must have a callback handler that deals with the `AuthorizeCallback`. This is used by the mechanism to determine whether the authenticated user is allowed to act on behalf of the requested authorization id, and also to obtain the canonicalized name of the authorized user (if canonicalization is applicable).

Most of the parameters, callbacks, and properties are described in the API documentation.

Debugging and Monitoring

The SunSASL and JdkSASL providers uses the Logging APIs to provide implementation logging output. This output can be controlled by using the logging configuration file and programmatic API (`java.util.logging`). The logger name used by the SunSASL provider

Copyright © 1993, 2026, Oracle and/or its affiliates.

is `javax.security.sasl`. Here is a sample logging configuration file that enables the `FINEST` logging level for the SunSASL provider:

```
javax.security.sasl.level=FINEST
handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=FINEST
```

Table 10-7 shows the mechanisms and the logging output that they generate:

Table 10-7 Logging Output

Mechanism	Logging Level	Information Logged
CRAM-MD5	`FINE`	Configuration properties; challenge/response messages
DIGEST-MD5	`INFO`	Message discarded due to encoding problem (for example, unmatched MACs, incorrect padding)
DIGEST-MD5	`FINE`	Configuration properties; challenge/response messages
DIGEST-MD5	`FINER`	More detailed information about challenge/response messages
DIGEST-MD5	`FINEST`	Buffers exchanged at the security layer
GSSAPI	`FINE`	Configuration properties; challenge/response messages
GSSAPI	`FINER`	More detailed information about challenge/response messages
GSSAPI	`FINEST`	Buffers exchanged at the security layer

Implementing a SASL Security Provider

There are three basic steps in implementing a SASL security provider:

1. Write a class that implements the `SaslClient` or `SaslServer` interface.

   This involves providing an implementation for the SASL mechanism. To implement a client mechanism, you need to implement the methods declared in the `SaslClient` interface. Similarly, for a server mechanism, you need to implement the methods declared in the `SaslServer` interface. For the purposes of this discussion, suppose you are developing an implementation for the client mechanism "SAMPLE-MECH", implemented by the class, `com.example.SampleMechClient`. You must decide what input are needed by the mechanism and how the implementation is going to collect them. For example, if the mechanism is username/password-based, then the implementation would likely need to collect that information via the callback handler parameter.

2. Write a factory class (that implements `SaslClientFactory` or `SaslServerFactory`) that creates instances of the class.

   This involves providing a factory class that will create instances of `com.example.SampleMechClient`. The factory needs to determine the characteristics of the mechanism that it supports (as described by the `Sasl.POLICY_*` properties) so that it can return an instance of the mechanism when the API user requests it using compatible policy properties. The factory may also check for validity of the parameters before creating the mechanism. For the purposes of this discussion, suppose the factory class is named

Copyright © 1993, 2026, Oracle and/or its affiliates.

`com.example.MySampleClientFactory`. Although our sample factory is responsible for only one mechanism, a single factory can be responsible for any number of mechanisms.

3. Write a JCA provider that registers the factory.

   This involves creating a JCA provider. The steps for creating a JCA provider is described in detail in Steps to Implement and Integrate a Provider. SASL client factories are registered using property names of the form `SaslClientFactory.`*mechName* while SASL server factories are registered using property names of the form `SaslServerFactory.`*mechName*

   mechName is the SASL mechanism's name. This is what's returned by `SaslClient.getMechanismName()` and `SaslServer.getMechanismName()`. Continuing with our example, here is how the provider would register the "SAMPLE-MECH" mechanism.

   ```
       put("SaslClientFactory.SAMPLE-MECH",
   "com.example.MySampleClientFactory");
   ```

   A single SASL provider might be responsible for many mechanisms. Therefore, it might have many invocations of `put` to register the relevant factories. The completed SASL provider can then be made available to applications using the instructions described in How SASL Mechanisms are Installed and Selected.

Copyright © 1993, 2026, Oracle and/or its affiliates.

11

XML Digital Signature API Overview and Tutorial

The Java XML Digital Signature API is a standard Java API for generating and validating XML Signatures. This API was defined under the Java Community Process as JSR 105.

XML Signatures can be applied to data of any type, XML or binary (see XML Signature Syntax and Processing). The resulting signature is represented in XML. An XML Signature can be used to secure your data and provide data integrity, message authentication, and signer authentication.

After providing a brief overview of XML Signatures and the XML Digital Signature API, this document presents two examples that demonstrate how to use the API to validate and generate an XML Signature. This document assumes that you have a basic knowledge of cryptography and digital signatures.

The API is designed to support all of the required or recommended features of the W3C Recommendation for XML-Signature Syntax and Processing. The API is extensible and pluggable and is based on the Java Cryptography Service Provider Architecture; see Java Cryptography Architecture (JCA) Reference Guide. The API is designed for two types of developers:

- Developers who want to use the XML Digital Signature API to generate and validate XML signatures
- Developers who want to create a concrete implementation of the XML Digital Signature API and register it as a cryptographic service of a JCA provider (see The Provider Class).

Package Hierarchy

The following six packages, which are contained in the `java.xml.crypto` module, comprise the XML Digital Signature API:

- `javax.xml.crypto`
- `javax.xml.crypto.dsig`
- `javax.xml.crypto.dsig.keyinfo`
- `javax.xml.crypto.dsig.spec`
- `javax.xml.crypto.dom`
- `javax.xml.crypto.dsig.dom`

The `javax.xml.crypto` package contains common classes that are used to perform XML cryptographic operations, such as generating an XML signature or encrypting XML data. Two notable classes in this package are the `KeySelector` class, which allows developers to supply implementations that locate and optionally validate keys using the information contained in a `KeyInfo` object, and the `URIDereferencer` class, which allows developers to create and specify their own URI dereferencing implementations.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The `javax.xml.crypto.dsig` package includes interfaces that represent the core elements defined in the W3C XML digital signature specification. Of primary significance is the `XMLSignature` class, which allows you to sign and validate an XML digital signature. Most of the XML signature structures or elements are represented by a corresponding interface (except for the KeyInfo structures, which are included in their own package and are discussed in the next paragraph). These interfaces include: `SignedInfo`, `CanonicalizationMethod`, `SignatureMethod`, `Reference`, `Transform`, `DigestMethod`, `XMLObject`, `Manifest`, `SignatureProperty`, and `SignatureProperties`. The `XMLSignatureFactory` class is an abstract factory that is used to create objects that implement these interfaces.

The `javax.xml.crypto.dsig.keyinfo` package contains interfaces that represent most of the `KeyInfo` structures defined in the W3C XML digital signature recommendation, including `KeyInfo`, `KeyName`, `KeyValue`, `X509Data`, `X509IssuerSerial`, `RetrievalMethod`, and `PGPData`. The `KeyInfoFactory` class is an abstract factory that is used to create objects that implement these interfaces.

The `javax.xml.crypto.dsig.spec` package contains interfaces and classes representing input parameters for the digest, signature, transform, or canonicalization algorithms used in the processing of XML signatures.

Finally, the `javax.xml.crypto.dom` and `javax.xml.crypto.dsig.dom` packages contains DOM-specific classes for the `javax.xml.crypto` and `javax.xml.crypto.dsig` packages, respectively. Only developers and users who are creating or using a DOM-based `XMLSignatureFactory` or `KeyInfoFactory` implementation will need to make direct use of these packages.

Service Providers

A Java XML Signature is a concrete implementation of the abstract `XMLSignatureFactory` and `KeyInfoFactory` classes and is responsible for creating objects and algorithms that parse, generate and validate XML Signatures and `KeyInfo` structures. A concrete implementation of `XMLSignatureFactory` must provide support for each of the required algorithms as specified by the W3C recommendation for XML Signatures. It can optionally support other algorithms as defined by the W3C recommendation or other specifications.

The Java XML Digital Signature API leverages the JCA provider model for registering and loading `XMLSignatureFactory` and `KeyInfoFactory` implementations.

Each concrete `XMLSignatureFactory` or `KeyInfoFactory` implementation supports a specific XML mechanism type that identifies the XML processing mechanism that an implementation uses internally to parse and generate XML signature and `KeyInfo` structures. This JSR supports one standard type, DOM. The XML Digital Signature provider implementation that is bundled with the JDK supports the DOM mechanism.

An XML Digital Signature API implementation should use underlying JCA engine classes, such as `java.security.Signature` and `java.security.MessageDigest`, to perform cryptographic operations.

In addition to the `XMLSignatureFactory` and `KeyInfoFactory` classes, JSR 105 supports a service provider interface for transform and canonicalization algorithms. The `TransformService` class allows you to develop and plug in an implementation of a specific transform or canonicalization algorithm for a particular XML mechanism type. The `TransformService` class uses the standard JCA provider model for registering and loading implementations. Each JSR 105 implementation should use the `TransformService` class to find a provider that supports transform and canonicalization algorithms in XML Signatures that it is generating or validating.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Introduction to XML Signatures

You can use an XML Signature to sign any arbitrary data, whether it is XML or binary. The data is identified via URIs in one or more Reference elements. XML Signatures are described in one or more of three forms: detached, enveloping, or enveloped. A detached signature is over data that is external, or outside of the signature element itself. Enveloping signatures are signatures over data that is inside the signature element, and an enveloped signature is a signature that is contained inside the data that it is signing.

Example of an XML Signature

The easiest way to describe the contents of an XML Signature is to show an actual sample and describe each component in more detail. Example 11-3 is an enveloped XML Signature generated over the contents of an XML document. The root element, `Envelop`, contains a `Signature` element:

```
<Envelope xmlns="urn:envelope">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <!-- ... -->
  </Signature>
</Envelope>
```

This `Signature` element has been inserted inside the content that it is signing, thereby making it an enveloped signature. The required `SignedInfo` element contains the information that is actually signed:

```
<Envelope xmlns="urn:envelope">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-
c14n-20010315#WithComments"/>
      <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-
sha256"/>
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature"/>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
        <DigestValue>/juoQ4bDxElf1M+KJauO20euW+QAvvPP0nDCruCQooM=</
DigestValue>
      </Reference>
    </SignedInfo>
    <!-- ... -->
  </Signature>
</Envelope>
```

The required `CanonicalizationMethod` element defines the algorithm used to canonicalize the `SignedInfo` element before it is signed or validated. Canonicalization is the process of converting XML content to a canonical form, to take into account changes that can invalidate a

Copyright © 1993, 2026, Oracle and/or its affiliates.

signature over that data. Canonicalization is necessary due to the nature of XML and the way it is parsed by different processors and intermediaries, which can change the data such that the signature is no longer valid but the signed data is still logically equivalent.

The required `SignatureMethod` element defines the digital signature algorithm used to generate the signature, in this case RSA with SHA-256.

One or more `Reference` elements identify the data that is digested. Each `Reference` element identifies the data via a URI. In this example, the value of the URI is the empty String (""), which indicates the root of the document. The optional `Transforms` element contains a list of one or more `Transform` elements, each of which describes a transformation algorithm used to transform the data before it is digested. In this example, there is one Transform element for the enveloped transform algorithm. The enveloped transform is required for enveloped signatures so that the signature element itself is removed before calculating the signature value. The required `DigestMethod` element defines the algorithm used to digest the data, in this case SHA-256. Finally the required `DigestValue` element contains the actual base64-encoded digested value.

The required `SignatureValue` element contains the base64-encoded signature value of the signature over the `SignedInfo` element.

The optional `KeyInfo` element contains information about the key that is needed to validate the signature:

```
    <KeyInfo>
      <KeyValue>
        <RSAKeyValue>
          <Modulus>
9hSmAKw/4TTw/1l1u1pYzdFm6lOjRB/5NfdGWl/fB8iAa/tiK0f1u/VWoK6SMtogYgSDKqQThbAu
9dy9rRnOWRGY2He1JtpOvGh0WCmIFUEs2P22HvEf+JGKVEpkoP4hv53ucT69T+7nKGK3/bjxgp+T
C7fbnVj651+jAHuDFlC8Txt1R8ZymfN5cUeHIH96dvNFrtai/uwZDbVMfhV9chL//+Vyhx4O5nHv
jfS+0So9Qi52YAbEyLu6+BLdu8wnMWapC88CfXsRwrpx8b6aCU0e6QSZyOvdgXWz3+9ifVTBDIxE
kjhL5OASx0qjvc+dPUOMvq7fJE05RRZLyb0YJw==
          </Modulus>
          <Exponent>AQAB</Exponent>
        </RSAKeyValue>
      </KeyValue>
    </KeyInfo>
```

This `KeyInfo` element contains a `KeyValue` element, which in turn contains a `RSAKeyValue` element consisting of the public key needed to validate the signature. `KeyInfo` can contain various content such as X.509 certificates and PGP key identifiers. See The `KeyInfo` Element in XML Signature Syntax and Processing for more information on the different `KeyInfo` types.

XML Signature Secure Validation Mode

The XML Signature secure validation mode can protect you from XML Signatures that may contain potentially hostile constructs that can cause denial-of-service or other types of security issues.

XML Signature secure validation mode is enabled by default.

If necessary, and at your own risk, you can disable the XML Signature secure validation mode by setting the system property `org.jcp.xml.dsig.secureValidation` to `false` or by setting the `org.jcp.xml.dsig.secureValidation` property to `Boolean.FALSE` with the

Copyright © 1993, 2026, Oracle and/or its affiliates.

`DOMValidateContext.setProperty()` method. The system property supersedes the value of the API property.

When XML Signature secure validation mode is enabled, XML Signatures are processed more securely. Limits are set on various XML Signature constructs to avoid conditions such as denial-of-service attacks. By default, it enforces the following restrictions:

- Forbids the use of XSLT transforms
- Restricts the number of `SignedInfo` or `Manifest` `Reference` elements to 30 or less
- Restricts the number of `Reference` transforms to 5 or less
- Forbids the use of MD5 or SHA-1 signatures or MD5 MAC algorithms
- Ensures that `Reference` IDs are unique to help prevent signature wrapping attacks
- Forbids `Reference` URIs of type `http`, `https`, or `file`
- Does not allow a `RetrievalMethod` element to reference another `RetrievalMethod` element
- Forbids RSA or DSA keys less than 1024 bits
- Forbids EC keys less than 224 bits

In addition, you can use the `jdk.xml.dsig.secureValidationPolicy` Security Property to control and fine-tune the restrictions listed previously or add additional restrictions. See the definition of this Security Property in the `java.security` file for more information.

XML Digital Signature API Examples

The following sections describe two examples that show how to use the XML Digital Signature API:

Validate Example

To compile and run the example, execute the following commands:

```
$ javac Validate.java
$ java Validate signature.xml
```

The sample program will validate the signature in the file `signature.xml` in the current working directory.

Example 11-1 Validate.java

```
import javax.xml.crypto.*;
import javax.xml.crypto.dsig.*;
import javax.xml.crypto.dom.*;
import javax.xml.crypto.dsig.dom.DOMValidateContext;
import javax.xml.crypto.dsig.keyinfo.*;
import java.io.FileInputStream;
import java.security.*;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import javax.xml.parsers.DocumentBuilderFactory;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;

/**
 * This is a simple example of validating an XML Signature using
 * the XML Signature API. It assumes the key needed to validate
 * the signature is contained in a KeyValue KeyInfo.
 */
public class Validate {

    //
    // Synopsis: java Validate [document]
    //
    //    where "document" is the name of a file containing the XML document
    //    to be validated.
    //
    public static void main(String[] args) throws Exception {

        // Instantiate the document to be validated
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setNamespaceAware(true);
        Document doc = null;
        try (FileInputStream fis = new FileInputStream(args[0])) {
            doc = dbf.newDocumentBuilder().parse(fis);
        }

        // Find Signature element
        NodeList nl =
            doc.getElementsByTagNameNS(XMLSignature.XMLNS, "Signature");
        if (nl.getLength() == 0) {
            throw new Exception("Cannot find Signature element");
        }

        // Create a DOM XMLSignatureFactory that will be used to unmarshal the
        // document containing the XMLSignature
        XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");

        // Create a DOMValidateContext and specify a KeyValue KeySelector
        // and document context
        DOMValidateContext valContext = new DOMValidateContext
            (new KeyValueKeySelector(), nl.item(0));

        // unmarshal the XMLSignature
        XMLSignature signature = fac.unmarshalXMLSignature(valContext);

        // Validate the generated XMLSignature
        boolean coreValidity = signature.validate(valContext);

        // Check core validation status
        if (coreValidity == false) {
            System.err.println("Signature failed core validation");
            boolean sv = signature.getSignatureValue().validate(valContext);
            System.out.println("signature validation status: " + sv);
            // check the validation status of each Reference
            Iterator<Reference> i =
                signature.getSignedInfo().getReferences().iterator();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                for (int j=0; i.hasNext(); j++) {
                    boolean refValid = i.next().validate(valContext);
                    System.out.println("ref["+j+"] validity status: " + refValid);
                }
            } else {
                System.out.println("Signature passed core validation");
            }
        }

        /**
         * KeySelector which retrieves the public key out of the
         * KeyValue element and returns it.
         * NOTE: If the key algorithm doesn't match signature algorithm,
         * then the public key will be ignored.
         */
        private static class KeyValueKeySelector extends KeySelector {
            public KeySelectorResult select(KeyInfo keyInfo,
                                            KeySelector.Purpose purpose,
                                            AlgorithmMethod method,
                                            XMLCryptoContext context)
                throws KeySelectorException {
                if (keyInfo == null) {
                    throw new KeySelectorException("Null KeyInfo object!");
                }
                SignatureMethod sm = (SignatureMethod) method;
                List<XMLStructure> list = keyInfo.getContent();

                for (int i = 0; i < list.size(); i++) {
                    XMLStructure xmlStructure = list.get(i);
                    if (xmlStructure instanceof KeyValue) {
                        PublicKey pk = null;
                        try {
                            pk = ((KeyValue)xmlStructure).getPublicKey();
                        } catch (KeyException ke) {
                            throw new KeySelectorException(ke);
                        }
                        // make sure algorithm is compatible with method
                        if (algEquals(sm.getAlgorithm(), pk.getAlgorithm())) {
                            return new SimpleKeySelectorResult(pk);
                        }
                    }
                }
                throw new KeySelectorException("No KeyValue element found!");
            }

            static boolean algEquals(String algURI, String algName) {
                if (algName.equalsIgnoreCase("DSA") &&
                    algURI.equalsIgnoreCase("http://www.w3.org/2009/xmldsig11#dsa-
sha256")) {
                    return true;
                } else if (algName.equalsIgnoreCase("RSA") &&
                           algURI.equalsIgnoreCase("http://www.w3.org/2001/04/
xmldsig-more#rsa-sha256")) {
                    return true;
                } else {
                    return false;
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
            }
        }
    }

    private static class SimpleKeySelectorResult implements KeySelectorResult
{
        private PublicKey pk;
        SimpleKeySelectorResult(PublicKey pk) {
            this.pk = pk;
        }

        public Key getKey() { return pk; }
    }
}
```

Example 11-2 envelope.xml

```
<Envelope xmlns="urn:envelope">
</Envelope>
```

Example 11-3 signature.xml

This file has been indented and formatted for readability.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Envelope xmlns="urn:envelope">
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-
c14n-20010315#WithComments"/>
      <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-
sha256"/>
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature"/>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
        <DigestValue>/juoQ4bDxElf1M+KJauO20euW+QAvvPP0nDCruCQooM=</
DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>
Vorr4nABCD7eWOjh4jn8pdM5iseGJPt4BmlgjEbxr05TsR9ObHq7WLVOBOtJfb3M6pXv6NnTucpH
e/97zHbuUMaNeGxCs/gN7YDUGOkQE1Gs4HAbGwXuTcif3pw+066ZW4uxyzapwS6lZHmqIm7PRl8I
NIQXVL4dezLe+rx77Kh+rZRheVe4UlTTP+TmIOaBZo93GQ5FudreMhSiuIC0Nx2SP7mAkt6+8kVH
luZouFbqriSvyhzIxDgyOXpm/PHCuuPU2scCokwjEZBtlZXDOl6lIWGllnyrptWntQ6F/ngQObI5
c2+npgCshq1svGuS/xx18MAFHGWi98Vj+07QCg==
    </SignatureValue>
    <KeyInfo>
      <KeyValue>
        <RSAKeyValue>
          <Modulus>
9hSmAKw/4TTw/1l1u1pYzdFm6lOjRB/5NfdGWl/fB8iAa/tiK0f1u/VWoK6SMtogYgSDKqQThbAu
9dy9rRnOWRGY2He1JtpOvGh0WCmIFUEs2P22HvEf+JGKVEpkoP4hv53ucT69T+7nKGK3/bjxgp+T
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
C7fbnVj651+jAHuDFlC8Txt1R8ZymfN5cUeHIH96dvNFrtai/uwZDbVMfhV9chL//+Vyhx4O5nHv
jfS+0So9Qi52YAbEyLu6+BLdu8wnMWapC88CfXsRwrpx8b6aCU0e6QSZyOvdgXWz3+9ifVTBDIxE
kjhL5OASx0qjvc+dPUOMvq7fJE05RRZLyb0YJw==
          </Modulus>
          <Exponent>AQAB</Exponent>
        </RSAKeyValue>
      </KeyValue>
    </KeyInfo>
  </Signature>
</Envelope>
```

Validating an XML Signature

This example shows you how to validate an XML Signature using the Java XML Digital Signature API. The example uses DOM (the Document Object Model) to parse an XML document containing a Signature element and a DOM implementation to validate the signature.

Instantiating the Document that Contains the Signature

First we use a JAXP `DocumentBuilderFactory` to parse the XML document containing the `Signature`. An application obtains the default implementation for `DocumentBuilderFactory` by calling the following line of code:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

We must also make the factory namespace-aware:

```
dbf.setNamespaceAware(true);
```

Next, we use the factory to get an instance of a `DocumentBuilder`, which is used to parse the document:

```
Document doc = null;
try (FileInputStream fis = new FileInputStream(args[0])) {
    doc = dbf.newDocumentBuilder().parse(fis);
}
```

Specifying the Signature Element to be Validated

We need to specify the `Signature` element that we want to validate, since there could be more than one in the document. We use the DOM method `Document.getElementsByTagNameNS`, passing it the XML Signature namespace URI and the tag name of the `Signature` element, as shown:

```
NodeList nl =
    doc.getElementsByTagNameNS(XMLSignature.XMLNS, "Signature");
if (nl.getLength() == 0) {
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        throw new Exception("Cannot find Signature element");
    }
```

This returns a list of all `Signature` elements in the document. In this example, there is only one `Signature` element.

Creating a Validation Context

We create an `XMLValidateContext` instance containing input parameters for validating the signature. Since we are using DOM, we instantiate a `DOMValidateContext` instance (a subclass of `XMLValidateContext`), and pass it two parameters, a `KeyValueKeySelector` object and a reference to the Signature element to be validated (which is the first entry of the `NodeList` we generated earlier):

```
DOMValidateContext valContext = new DOMValidateContext
    (new KeyValueKeySelector(), nl.item(0));
```

The `KeyValueKeySelector` is explained in greater detail in Using KeySelectors.

Unmarshalling the XML Signature

We extract the contents of the `Signature` element into an `XMLSignature` object. This process is called unmarshalling. The `Signature` element is unmarshalled using an `XMLSignatureFactory` object. An application can obtain a DOM implementation of `XMLSignatureFactory` by calling the following line of code:

```
XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
```

We then invoke the `unmarshalXMLSignature` method of the factory to unmarshal an `XMLSignature` object, and pass it the validation context we created earlier:

```
XMLSignature signature = fac.unmarshalXMLSignature(valContext);
```

Validating the XML Signature

Now we are ready to validate the signature. We do this by invoking the validate method on the `XMLSignature` object, and pass it the validation context as follows:

```
boolean coreValidity = signature.validate(valContext);
```

The validate method returns "true" if the signature validates successfully according to the core validation rules in the W3C XML Signature Recommendation, and false otherwise.

What If the XML Signature Fails to Validate?

If the `XMLSignature.validate` method returns false, we can try to narrow down the cause of the failure. There are two phases in core XML Signature validation:

- Signature validation (the cryptographic verification of the signature)

Copyright © 1993, 2026, Oracle and/or its affiliates.

- Reference validation (the verification of the digest of each reference in the signature)

Each phase must be successful for the signature to be valid. To check if the signature failed to cryptographically validate, we can check the status, as follows:

```
boolean sv = signature.getSignatureValue().validate(valContext);
System.out.println("signature validation status: " + sv);
```

We can also iterate over the references and check the validation status of each one, as follows:

```
Iterator<Reference> i =
    signature.getSignedInfo().getReferences().iterator();
for (int j=0; i.hasNext(); j++) {
    boolean refValid = i.next().validate(valContext);
    System.out.println("ref["+j+"] validity status: " + refValid);
}
```

Using KeySelectors

`KeySelectors` are used to find and select keys that are needed to validate an `XMLSignature`. Earlier, when we created a `DOMValidateContext` object, we passed a `KeyValueKeySelector` object as the first argument:

```
DOMValidateContext valContext = new DOMValidateContext
    (new KeyValueKeySelector(), nl.item(0));
```

Alternatively, we could have passed a `PublicKey` as the first argument if we already knew what key is needed to validate the signature. However, we often don't know.

The `KeyValueKeySelector` class is a concrete implementation of the abstract `KeySelector` class. The `KeyValueKeySelector` implementation tries to find an appropriate validation key using the data contained in `KeyValue` elements of the `KeyInfo` element of an `XMLSignature`. It does not determine if the key is trusted. This is a very simple `KeySelector` implementation, designed for illustration rather than real-world usage. A more practical example of a `KeySelector` is one that searches a `KeyStore` for trusted keys that match `X509Data` information (for example, `X509SubjectName`, `X509IssuerSerial`, `X509SKI`, or `X509Certificate` elements) contained in a `KeyInfo`.

The implementation of the `KeyValueKeySelector` class is as follows:

```
private static class KeyValueKeySelector extends KeySelector {
    public KeySelectorResult select(KeyInfo keyInfo,
                                    KeySelector.Purpose purpose,
                                    AlgorithmMethod method,
                                    XMLCryptoContext context)
        throws KeySelectorException {
        if (keyInfo == null) {
            throw new KeySelectorException("Null KeyInfo object!");
        }
        SignatureMethod sm = (SignatureMethod) method;
        List<XMLStructure> list = keyInfo.getContent();
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
                for (int i = 0; i < list.size(); i++) {
                    XMLStructure xmlStructure = list.get(i);
                    if (xmlStructure instanceof KeyValue) {
                        PublicKey pk = null;
                        try {
                            pk = ((KeyValue)xmlStructure).getPublicKey();
                        } catch (KeyException ke) {
                            throw new KeySelectorException(ke);
                        }
                        // make sure algorithm is compatible with method
                        if (algEquals(sm.getAlgorithm(), pk.getAlgorithm())) {
                            return new SimpleKeySelectorResult(pk);
                        }
                    }
                }
                throw new KeySelectorException("No KeyValue element found!");
            }

            static boolean algEquals(String algURI, String algName) {
                if (algName.equalsIgnoreCase("DSA") &&
                    algURI.equalsIgnoreCase("http://www.w3.org/2009/xmldsig11#dsa-
sha256")) {
                    return true;
                } else if (algName.equalsIgnoreCase("RSA") &&
                           algURI.equalsIgnoreCase("http://www.w3.org/2001/04/
xmldsig-more#rsa-sha256")) {
                    return true;
                } else {
                    return false;
                }
            }
        }

        private static class SimpleKeySelectorResult implements KeySelectorResult
{
            private PublicKey pk;
            SimpleKeySelectorResult(PublicKey pk) {
                this.pk = pk;
            }

            public Key getKey() { return pk; }
        }
```

GenEnveloped Example

To compile and run this sample, execute the following command:

```
$ javac GenEnveloped.java
$ java GenEnveloped envelope.xml envelopedSignature.xml
```

The sample program will generate an enveloped signature of the document in the file `envelope.xml` and store it in the file `envelopedSignature.xml` in the current working directory.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Example 11-4 GenEnveloped.java

```
import javax.xml.crypto.dsig.*;
import javax.xml.crypto.dsig.dom.DOMSignContext;
import javax.xml.crypto.dsig.keyinfo.*;
import javax.xml.crypto.dsig.spec.*;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.security.*;
import java.util.List;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;

/**
 * This is a simple example of generating an Enveloped XML
 * Signature using the Java XML Digital Signature API. The
 * resulting signature will look like (key and signature
 * values will be different):
 *
 * <pre><code>
 *<Envelope xmlns="urn:envelope">
 * <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
 *   <SignedInfo>
 *     <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-
c14n-20010315"/>
 *     <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-
sha256"/>
 *     <Reference URI="">
 *       <Transforms>
 *         <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature"/>
 *       </Transforms>
 *       <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
 *       <DigestValue>/
juoQ4bDxElf1M+KJauO20euW+QAvvPP0nDCruCQooM=<DigestValue>
 *     </Reference>
 *   </SignedInfo>
 *   <SignatureValue>
 *     YeS+F0uiYv0h946M69Q9pKFNnD6dxUwLA8QT3GX/0H3cSPKRnNFyZiR4RPgaA1ir/
ztb4rt6Lqb8
 *
hgwPERIa5qhoGUJyHDfUTcQ0Xqn1jYCVoC3ho+oUgJPXNVgtMAtpvOgxcWXUPATYdyimO6RrHF8+
 *
JXDkeICI9BPA4NKN1i77CAy6JJbaA87aNIpMJPImwJf8CM7mYsXremZz+RsafNE2cXXRzAoNOynC
 *     pi4oPYpE7CBLzhd23gf7zYRoyT06/
bVIj4j3qOlVY1TQofsQ20NtAz6PbqAs7QkNoDzkX1CYlDSJ
 *     U8cGHuwXpul/UIpOiL6MZF8I/YI4ZlJn+O8Mvg==
 *   </SignatureValue>
 *   <KeyInfo>
 *     <KeyValue>
 *       <RSAKeyValue>
 *         <Modulus>
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
 *           mH0S/
iw2K2tFTFHI75BtB67pzjR52HvQ8K7Xi5UX3NJm0oA+KX2mm0IrVcUuv609vbAAyQoW7CWm
 *           4kswVgStCm68dlw36309cxrEmPhG+PKBmUaGuBmRzwityjXRyRZJ6yaLenE8SJO/
DC5ntQvmHqQQ
 *
qeOJYvz2Cbi2bi6x9XwmpqOfZCE5iTvYwioEsrglhP1uLG9fiXyNR2PXUTyLqD91HLhZFj1CEiU7
 *           aE+
+WfkKaowIx5p8e3F6hQ+VFRNXjtemK5aajuL0gwU+Oujg9ijgbyMh19vBoI8LruJoMOBrYFNN
 *           2boQJ3wP0Ek7CPIqAzQB5MnmvKc9jICKiiZVZw==
 *         </Modulus>
 *         <Exponent>AQAB</Exponent>
 *       </RSAKeyValue>
 *     </KeyValue>
 *   </KeyInfo>
 * </Signature>
 *</Envelope>
 * </code></pre>
 */
public class GenEnveloped {

    //
    // Synopsis: java GenEnveloped [document] [output]
    //
    //    where "document" is the name of a file containing the XML document
    //    to be signed, and "output" is the name of the file to store the
    //    signed document. The 2nd argument is optional - if not specified,
    //    standard output will be used.
    //
    public static void main(String[] args) throws Exception {

        // Instantiate the document to be signed
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setNamespaceAware(true);
        Document doc = null;
        try (FileInputStream fis = new FileInputStream(args[0])) {
            doc = dbf.newDocumentBuilder().parse(fis);
        }

        // Create a RSA KeyPair
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(2048);
        KeyPair kp = kpg.generateKeyPair();

        // Create a DOMSignContext and specify the RSA PrivateKey and
        // location of the resulting XMLSignature's parent element
        DOMSignContext dsc = new DOMSignContext
            (kp.getPrivate(), doc.getDocumentElement());

        // Create a DOM XMLSignatureFactory that will be used to generate the
        // enveloped signature
        XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");

        // Create a Reference to the enveloped document (in this case we are
        // signing the whole document, so a URI of "" signifies that) and
        // also specify the SHA256 digest algorithm and the ENVELOPED
Transform.
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        Reference ref = fac.newReference
            ("", fac.newDigestMethod(DigestMethod.SHA256, null),
             List.of
              (fac.newTransform
                (Transform.ENVELOPED, (TransformParameterSpec) null)),
             null, null);

        // Create the SignedInfo
        SignedInfo si = fac.newSignedInfo
            (fac.newCanonicalizationMethod
             (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
              (C14NMethodParameterSpec) null),
             fac.newSignatureMethod("http://www.w3.org/2001/04/xmldsig-
more#rsa-sha256", null),
             List.of(ref));

        // Create a KeyValue containing the RSA PublicKey that was generated
        KeyInfoFactory kif = fac.getKeyInfoFactory();
        KeyValue kv = kif.newKeyValue(kp.getPublic());

        // Create a KeyInfo and add the KeyValue to it
        KeyInfo ki = kif.newKeyInfo(List.of(kv));

        // Create the XMLSignature (but don't sign it yet)
        XMLSignature signature = fac.newXMLSignature(si, ki);

        // Marshal, generate (and sign) the enveloped signature
        signature.sign(dsc);

        // output the resulting document
        OutputStream os;
        if (args.length > 1) {
           os = new FileOutputStream(args[1]);
        } else {
           os = System.out;
        }

        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer trans = tf.newTransformer();
        trans.transform(new DOMSource(doc), new StreamResult(os));
    }
}
```

Example 11-5 envelope.xml

```
<Envelope xmlns="urn:envelope">
</Envelope>
```

Generating an XML Signature

This example shows you how to generate an XML Signature using the XML Digital Signature API. More specifically, the example generates an enveloped XML Signature of an XML document. An enveloped signature is a signature that is contained inside the content that it is

Copyright © 1993, 2026, Oracle and/or its affiliates.

signing. The example uses DOM (the Document Object Model) to parse the XML document to be signed and a DOM implementation to generate the resulting signature.

A basic knowledge of XML Signatures and their different components is helpful for understanding this section. See XML Signature Syntax and Processing Version 1.1 for more information.

Instantiating the Document to be Signed

First, we use a JAXP `DocumentBuilderFactory` to parse the XML document that we want to sign. An application obtains the default implementation for `DocumentBuilderFactory` by calling the following line of code:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

We must also make the factory namespace-aware:

```
dbf.setNamespaceAware(true);
```

Next, we use the factory to get an instance of a `DocumentBuilder`, which is used to parse the document:

```
Document doc = null;
try (FileInputStream fis = new FileInputStream(args[0])) {
    doc = dbf.newDocumentBuilder().parse(fis);
}
```

Creating a Public Key Pair

We generate a public key pair. Later in the example, we will use the private key to generate the signature. We create the key pair with a `KeyPairGenerator`. In this example, we will create a RSA `KeyPair` with a length of 2048 bytes:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(2048);
KeyPair kp = kpg.generateKeyPair();
```

In practice, the private key is usually previously generated and stored in a `KeyStore` file with an associated public key certificate.

Creating a Signing Context

We create an `XMLSignContext` containing input parameters for generating the signature. Since we are using DOM, we instantiate a `DOMSignContext` (a subclass of `XMLSignContext`), and pass it two parameters, the private key that will be used to sign the document and the root of the document to be signed:

```
DOMSignContext dsc = new DOMSignContext
    (kp.getPrivate(), doc.getDocumentElement());
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> You can use your own instance of `SecureRandom` when creating a `DOMSignContext`. Specify your `SecureRandom` instance with the property `jdk.xmldsig.SecureRandom` by calling `setProperty` as follows, where `dsc` is your `DOMSignContext`:
>
> ```
> dsc.setProperty("jdk.xmldsig.SecureRandom", mySecureRandom)
> ```
>
> Call `setProperty` before calling `XMLSignature.sign(XMLSignContext)`. The `sign` method is described in Generating an XML Signature.

Assembling the XML Signature

We assemble the different parts of the Signature element into an `XMLSignature` object. These objects are all created and assembled using an `XMLSignatureFactory` object. An application obtains a DOM implementation of `XMLSignatureFactory` by calling the following line of code:

```
XMLSignatureFactory fac = XMLSignatureFactory.getInstance("DOM");
```

We then invoke various factory methods to create the different parts of the `XMLSignature` object. We create a `Reference` object, passing to it the following:

- The URI of the object to be signed (We specify a URI of "", which implies the root of the document.)
- The `DigestMethod` (we use SHA256)
- A single `Transform`, the enveloped `Transform`, which is required for enveloped signatures so that the signature itself is removed before calculating the signature value

```
Reference ref = fac.newReference
    ("", fac.newDigestMethod(DigestMethod.SHA256, null),
     List.of
      (fac.newTransform
        (Transform.ENVELOPED, (TransformParameterSpec) null)),
     null, null);
```

Next, we create the `SignedInfo` object, which is the object that is actually signed. When creating the `SignedInfo`, we pass as parameters:

- The `CanonicalizationMethod` (we use inclusive and preserve comments)
- The `SignatureMethod` (we use RSA)
- A list of `References` (in this case, only one)

```
SignedInfo si = fac.newSignedInfo
    (fac.newCanonicalizationMethod
     (CanonicalizationMethod.INCLUSIVE_WITH_COMMENTS,
      (C14NMethodParameterSpec) null),
     fac.newSignatureMethod("http://www.w3.org/2001/04/xmldsig-more#rsa-
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
sha256", null),
         List.of(ref));
```

Next, we create the optional `KeyInfo` object, which contains information that enables the recipient to find the key needed to validate the signature. In this example, we add a `KeyValue` object containing the public key. To create `KeyInfo` and its various subtypes, we use a `KeyInfoFactory` object, which can be obtained by invoking the `getKeyInfoFactory` method of the *XMLSignatureFactory*, as follows:

```
KeyInfoFactory kif = fac.getKeyInfoFactory();
```

We then use the `KeyInfoFactory` to create the `KeyValue` object and add it to a `KeyInfo` object:

```
KeyValue kv = kif.newKeyValue(kp.getPublic());
KeyInfo ki = kif.newKeyInfo(List.of(kv));
```

Finally, we create the `XMLSignature` object, passing as parameters the `SignedInfo` and `KeyInfo` objects that we created earlier:

```
XMLSignature signature = fac.newXMLSignature(si, ki);
```

Notice that we haven't actually generated the signature yet; we'll do that in the next step.

Generating the XML Signature

Now we are ready to generate the signature, which we do by invoking the sign method on the `XMLSignature` object, and pass it the signing context as follows:

```
signature.sign(dsc);
```

The resulting document now contains a signature, which has been inserted as the last child element of the root element.

Printing or Displaying the Resulting Document

You can use the following code to print the resulting signed document to a file or standard output:

```
OutputStream os;
if (args.length > 1) {
   os = new FileOutputStream(args[1]);
} else {
   os = System.out;
}

TransformerFactory tf = TransformerFactory.newInstance();
Transformer trans = tf.newTransformer();
trans.transform(new DOMSource(doc), new StreamResult(os));
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Copyright © 1993, 2026, Oracle and/or its affiliates.

12

Java API for XML Processing (JAXP) Security Guide

The JDK and Java XML APIs have been improved over the years with various measures and tools that can help prevent applications from being exploited by XML-related attacks. This guide shows you how to use the secure processing features of Java API for XML Processing (JAXP) to safeguard your applications and systems.

See the `java.xml` module summary for more information.

Security Issues in XML Processing

XML processing can expose applications to certain vulnerabilities. Among the most prominent and well-known attacks are the XML External Entity (XXE) injection attack and the exponential entity expansion attack, also know as the XML bomb or billion laughs attack. These attacks can potentially cause serious damage to a system by denying its services or worse, lead to the loss of sensitive data.

You should evaluate your applications' requirements and operating environment to assess the level of potential threat, for example, whether or to what extent the applications are exposed to untrusted XML sources.

XML External Entity Injection Attack

The XML, XML Schema, and XSLT standards define a number of structures that enable the embedding of external content in XML documents through system identifiers that reference external resources. In general, XML processors resolve and retrieve almost all of these external resources; see External Resources Supported by XML, Schema, and XSLT Standards for a list of constructs that support the inclusion of external resources. In addition, some constructs enable the execution of applications through external functions. XML External Entity (XXE) injection attacks exploit XML processors that have not been secured by restricting the external resources that it may resolve, retrieve, or execute. This can result in disclosing sensitive data such as passwords or enabling arbitrary execution of code.

External Resources Supported by XML, Schema, and XSLT Standards

XML, Schema, and XSLT standards support the following constructs that require external resources. The default behavior of the JDK XML processors is to make a connection and fetch the external resources as specified.

- External DTD: references an external Document Type Definition (DTD), for example:

  ```
  <!DOCTYPE root_element SYSTEM "url">
  ```

- External Entity Reference: Refers to external data, the following is the syntax:

  ```
  <!ENTITY name SYSTEM "url">
  ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

- General entity reference, for example:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE doc [<!ENTITY otherFile SYSTEM "otherFile.xml">]>
<doc>
    <a>
        <b>&otherFile;</b>
    </a>
</doc>
```

- External Parameter Entities: The following is the syntax:

```
<!ENTITY % name SYSTEM uri>
```

  The following is an example:

```
<?xml version="1.0" standalone="no"?>
    <!DOCTYPE doc [
      <!ENTITY % ent1 SYSTEM "http://www.example.com/student.dtd">
      %ent1;
    ]>
```

- XInclude: Includes an external infoset in an XML document, for example:

```
<Book xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include href=toc.xml"/>
    <xi:include href=part1.xml"/>
    <xi:include href=part2.xml"/>
    <xi:include href=index.xml"/>
</Book>
```

- References to XML Schema components using the `schemaLocation` attribute and `import` and `include` elements, for example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:include schemaLocation="http://www.example.com/schema/
schema1.xsd"/>
    <!-- ... -->
</xs:schema>
```

- Combining style sheets using import or include elements, the following is the syntax:

```
<xsl:include href="include.xsl"/>
```

- `xml-stylesheet` processing instruction: Used to include a stylesheet in an XML document, for example:

```
<?xml-stylesheet href="include.xsl" type="text/xsl"?>
```

- XSLT `document()` function: Used to access nodes in an external XML document, for example:

```
<xsl:variable name="dummy" select="document('DocumentFunc2.xml')"/>
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

Exponential Entity Expansion Attack

The exponential entity expansion attack, also know as the XML bomb or billion laughs attack, is a denial-of-service attack that involves XML parsers. The basic exploit is to have several layers of nested entities, each referring to a number of entities of the next layer. The following is a sample SOAP document that contains deeply nested entity references:

```
<?xml version="1.0" encoding ="UTF-8"?>
<!DOCTYPE bbb[
    <!ENTITY x100 "bbb">
    <!ENTITY  x99 "&x100;&x100;">
    <!ENTITY  x98 "&x99;&x99;">
    ...
    <!ENTITY   x2 "&x3;&x3;">
    <!ENTITY   x1 "&x2;&x2;">
]>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=...>
    <SOAP-ENV:Body>
        <ns1:aaa xmlns:ns1="urn:aaa" SOAP-ENV:encodingStyle="...">
            <foobar xsi:type="xsd:string">&x1;</foobar>
        </ns1:aaa>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

When an XML parser encounters such a document, it will attempt to resolve the entity declaration by expanding the references. Because the references are nested, the expansion becomes exponential by the number of entities each refers to. Such a process can lead the XML parser to consume 100% of CPU time and a large amount of memory, and eventually the system runs out of memory.

Configuring JAXP for Secure XML Processing

JAXP consists of a set of APIs for parsing, serializing, transforming, querying, and traversing XML documents. See JAXP APIs.

JAXP adds a layer of security through factories and processors. You access APIs from JAXP through factories, which ensure that they are configured according to JAXP properties that you've set based on your security requirements. You configure and instantiate factories with their respective processors. See Factories and Processors.

You configure JAXP processors through JAXP properties. You can set JAXP properties through JAXP factories, as System properties, or in a JAXP configuration file. See Configuring with JAXP Properties.

The section Security-Related Properties describes properties you can set to configure JAXP for secure XML processing.

The section Composite Processors describes how enabling FSP for a composite processor such as a validator or transformer also enables FSP for any internal parser that the composite processor uses.

In addition, JAXP enables you to register custom resolvers and catalogs, which intercept references to external resources and resolve them with local ones. This eliminates the need to

Copyright © 1993, 2026, Oracle and/or its affiliates.

read and access external resources, which helps remove a source of potential risk. See Using Resolvers and Catalogs.

Secure XML Processing Best Practices

The JDK supports JAXP `FEATURE_SECURE_PROCESSING` (FSP) (see The FEATURE_SECURE_PROCESSING Security Directive), which enables a baseline of secure XML processing by imposing key restrictions through various features and properties (see Configuring with JAXP Properties). When FSP is enabled on XML factories by default or by explicit settings, the JDK applies restrictive values to relevant security properties. You can also directly set these properties through API calls or system properties. You can also centrally manage them in the JAXP Configuration File (JCF) (see Setting JAXP Properties in a JAXP Configuration File).

Key Security Configurations

- **FEATURE_SECURE_PROCESSING (FSP)**

  Securing your XML applications starts with enabling the FSP directive on XML factories (such as DOM, SAX, validation, transformation, and XPath). See The FEATURE_SECURE_PROCESSING Security Directive. While the JDK applies most secure defaults, it does not restrict external connections by default, so explicit configuration is necessary. The following code, for example, sets FSP on a DOM factory:

  ```
  DocumentBuilderFactory factory =
  DocumentBuilderFactory.newDefaultNSInstance();
  factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
  ```

  The explicit FSP setting enforces resource limits (see JAXP Properties for Processing Limits), disables external entity resolution (see External Access Properties), prevents use of third-party parser overrides (see JAXP Property for Third-Party Parsers), and restricts XSLT extension functions (see JAXP Property for Extension Functions).

- **JAXP Configuration File (JCF)**

  You can set security-related JAXP properties (see Configuring with JAXP Properties) programmatically, through system properties, or by using the JAXP Configuration File (JCF) (see Setting JAXP Properties in a JAXP Configuration File). Using the JCF provides a convenient, code-free means of managing security settings across all XML processor instances in the JVM.

- **External Access Restrictions**

  Preventing arbitrary external access is essential for XML security. This can be achieved by setting FSP on XML factories or, where more granular control is necessary, by configuring External Access Properties (EAP) (see External Access Properties) or the built-in catalog's resolve property `jdk.xml.jdkcatalog.resolve` (see The Built-In Catalog). Note that for the StAX processor (see the `javax.xml.stream` package), FSP is not supported, so EAP or resolve settings must be used.

- **Resolvers and Catalogs**

  Restricting all external access may be too limiting for some applications. To safely handle external resources, implement resolvers or catalogs (see Using Resolvers and Catalogs). These mechanisms allow fine-grained control over how external references are resolved while blocking unwanted connections.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> While FSP is enabled by default in the JDK, the default configuration does not restrict external entity resolution. For comprehensive security, explicitly set FSP through the API and configure external access restrictions as needed.

JAXP APIs

JAXP consists of a set of APIs built XML technologies and standards that are essential for XML processing. These include APIs for the following:

- Parsing: JAXP Parsing API (`javax.xml.parsers` package, which is based on the Document Object Model (DOM) (`org.w3c.dom` package) and Simple API for XML Parsing (SAX) (`org.xml.sax` package)
- Streaming API for XML (StAX) (`javax.xml.stream` package)
- Serializing: StAX and Extensible Stylesheet Language Transformations (XSLT) (`javax.xml.transform` package)
- Transformation: JAXP Transformation API (`javax.xml.transform` package) and XSLT (Extensible Stylesheet Language Transformations)
- Querying and traversing XML documents: XML Path Language (XPath) API (`javax.xml.xpath` package)
- Resolving external resources: XML Catalog API (`javax.xml.catalog` package)

Factories and Processors

Factories are the entry points of each JAXP API. They provide methods that enable applications to set JAXP properties programmatically before creating processors. Factories also support the JAXP lookup mechanism, which enable you to deploy applications with third-party implementations instead of JDK implementations.

Processors are aggregates of parsers (or readers), serializers (or writers), validators, and transformers that control and perform the processing in their respective areas. Factories configure and instantiate their corresponding factories. For example, you configure and instantiate the `DocumentBuilder` and `SAXParser` processors with the `DocumentBuilderFactory` and `SAXParserFactory` factories, respectively.

Configuring with JAXP Properties

You configure JAXP processors through JAXP properties. The following lists the ways you can set JAXP properties in order of precedence from highest to lowest. For example, if you set a JAXP property's value through a factory, then it will override any other value set in a different way.

- Setting JAXP Properties Through JAXP Factories: You can set a JAXP property through a JAXP factory in your code.
- Setting JAXP Properties as System Properties: You can set a JAXP property like a System property by setting its value on the command line with the following syntax:

  ```
  -D<property name>=<property value>
  ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

You can also set a System property's value with the method `System.setProperty(String key, String name)` or `System.setProperties(Properties)`.

- Setting JAXP Properties in a JAXP Configuration File: You can set a JAXP property in a user-defined JAXP configuration file or in the default JAXP configuration file, *<java-home>*`/conf/jaxp.properties` .

Not all JAXP properties can be set by all of these ways. See the `java.xml` module summary to determine which method you can set a specific JAXP property.

If you haven't set a particular JAXP property with one of these ways, then the Java runtime uses the value specified in the default JAXP properties file. If a property doesn't exist in this file, then the Java runtime uses its default value. However, if `FEATURE_SECURE_PROCESSING` (FSP) is turned on, then the Java runtime uses a more restrictive value, if applicable. See the table Implementation Specific Properties in the `java.xml` module summary for more information.

See the default JAXP configuration file *<java-home>*`/conf/jaxp.properties` for information about additional JAXP properties.

Setting JAXP Properties Through JAXP Factories

If you can modify your application's code, or you're creating a new application, then you can set JAXP properties through JAXP factories or a parser. Set these properties through the following interfaces:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setAttribute(name, value);

SAXParserFactory spf = SAXParserFactory.newInstance();
SAXParser parser = spf.newSAXParser();
parser.setProperty(name, value);

SchemaFactory schemaFactory = SchemaFactory.newInstance(schemaLanguage);
schemaFactory.setProperty(name, value);

TransformerFactory factory = TransformerFactory.newInstance();
factory.setAttribute(name, value);

XMLInputFactory xif = XMLInputFactory.newInstance();
xif.setProperty(name, value);

XPathFactory xf = XPathFactory.newInstance();
xf.setProperty(name, value);
```

The following is an example of setting processing limits:

```
dbf.setAttribute("jdk.xml.entityExpansionLimit", "2000");
dbf.setAttribute("jdk.xml.totalEntitySizeLimit", "100000");
dbf.setAttribute("jdk.xml.maxParameterEntitySizeLimit", "10000");
dbf.setAttribute("jdk.xml.maxElementDepth", "100");

factory.setAttribute("jdk.xml.xpathTotalOpLimit", "1000");
xf.setProperty("jdk.xml.xpathExprGrpLimit", "20");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The following is an example of limiting a DOM parser to only local connections for external DTDs:

```
dbf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "file, jar:file");
```

If a parser module within the application handles untrusted sources, it may further restrict access. The following code overrides those in the JAXP configuration file and those specified by System properties and enables the XML processor to read local files only:

```
DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
dbf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "file");
// ...
SchemaFactory schemaFactory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
schemaFactory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "file");
schemaFactory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "file");
```

As described in Scope and Order of Setting JAXP Properties, JAXP properties specified through JAXP factories have the narrowest scope, affecting only the processors created by the factories, and therefore override any default settings, System properties, and those in the JAXP configuration file. By setting JAXP properties through JAXP factories, you can ensure that your applications behave the same way regardless of which JDK release you're using or whether JAXP properties are set through other means.

Setting JAXP Properties as System Properties

System properties may be useful if you can't modify your application's code.

To set JAXP properties for an entire JDK invocation, set their corresponding System properties on the command line. For example, if your application, `MyApp` requires access to external DTDs and schemas, then set the System properties `javax.xml.accessExternalDTD` and `javax.xml.accessExternalSchema` as follows:

```
java -Djavax.xml.accessExternalDTD="file,http" -
Djavax.xml.accessExternalSchema="file, http" MyApp
```

To set JAXP properties for only a portion of the application, set their corresponding System properties before the portion, and then clear them afterward. For example, if your application requires access to external DTDs and schemas, then add these lines to your application's initialization code block:

```
System.setProperty("javax.xml.accessExternalDTD", "file, http");
System.setProperty("javax.xml.accessExternalSchema", "file, http");
```

Then, once your application is done processing XML documents or before it exits, clear out the properties as follows:

```
System.clearProperty("javax.xml.accessExternalDTD");
System.clearProperty("javax.xml.accessExternalSchema");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> JAXP properties related to processing limits are specific to the JDK implementation while those related to external access propeties (EAPs; see External Access Properties in the JavaDoc API documentation for `XMLConstants`) are standard properties.

Setting JAXP Properties in a JAXP Configuration File

You can set JAXP properties in a JAXP configuration file, which is a `java.util.Properties` file. (See Properties in *The Java Tutorials* for more information about Java properties.)

Topics

- Format of a JAXP Configuration File
- Specifying a User-Defined JAXP Configuration File
- Default JAXP Configuration File
- JAXP Configuration Template with Strict Settings

Format of a JAXP Configuration File

A JAXP configuration file is a text file that contains names and values of JAXP properties, one name-value pair on each line. For example, the following JAXP configuration file sets the `maxGeneralEntitySizeLimit` processing limit property to 2000 and restricts access to the file and HTTP protocols for external references set by the `stylesheet` processing instruction, `document` function, and the `import` and `include` elements.

```
jdk.xml.maxGeneralEntitySizeLimit=2000
javax.xml.accessExternalStylesheet=file, http
```

If you don't want to allow any external connection by XML processors, you can set all access external restrictions to `file` only:

```
javax.xml.accessExternalDTD=file
javax.xml.accessExternalSchema=file
javax.xml.accessExternalStylesheet=file
```

If you want to prevent applications from accidentally reading external files through an XML processor, set the external access properties (EAPs) as in the following JAXP configuration file:

```
javax.xml.accessExternalDTD=""
javax.xml.accessExternalSchema=""
javax.xml.accessExternalStylesheet=""
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Tip**
>
> If you can specify a JAXP property as a System property, then you can specify the same JAXP property in a JAXP configuration file. See the `java.xml` module summary to determine where to set the value of a specific JAXP property.

Specifying a User-Defined JAXP Configuration File

To use your own JAXP configuration file, specify its name and location with the `java.xml.config.file` System property. If you specify a file path that's not absolute, then it's relative to the current working directory. If the value of this System property hasn't been specified when a JAXP factory is instantiated, then no further attempt will be made to check its value.

A user-defined JAXP configuration file is read only once when the JAXP implementation is initialized. This means that it won't work if you modify it after a JAXP factory has been instantiated.

Default JAXP Configuration File

The JDK contains a default JAXP configuration file *<java_home>*`/conf/jaxp.properties`. However, it's recommended that you create your own user-defined JAXP configuration file instead.

If you have specified a user-defined JAXP configuration file with the System property `java.xml.config.file` and that file exists, then any settings in it override those in the default JAXP configuration file.

JAXP Configuration Template with Strict Settings

The JAXP configuration template can help you validate that your applications are prepared for future JDK releases. You can use the template, which specifies more restrictive XML processing settings, to identify issues such as a processor unknowingly making outbound network connections to fetch DTDs or an XML processor that relies on extension functions.

Follow these steps to test the readiness of your application with the JAXP configuration template:

1. Copy the JAXP configuration template, *<java_home>*`/conf/jaxp-strict.properties.template`, to a new file with the `.properties` extension. For example, the following command copies the template to a file named `/path/to/jaxp-strict.properties`:

   ```
   cp <java_home>/conf/jaxp-strict.properties.template /path/to/jaxp-strict.properties
   ```

2. Run your application, specifying the system property `java.xml.config.file` with the path where you copied the JAXP configuration file template to override the default configuration. For example:

   ```
   java -Djava.xml.config.file=/path/to/jaxp-strict.properties myApp
   ```

If you decide that your applications are ready for a more secure configuration, then you can use the configuration template as your JAXP configuration file.

See the section Configuration File in the `java.xml` module JavaDoc API documentation for more information.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Scope and Order of Setting JAXP Properties

JAXP properties can be set in multiple ways, including through API methods, in system properties, and in a JAXP configuration file. When not explicitly set, they will be initialized with default values or more restrictive values when `FEATURE_SECURE_PROCESSING` (FSP) is enabled. The order of precedence for properties is as follows, from highest to lowest:

- The APIs for factories or processors
- System properties
- User-defined JAXP configuration file
- The default JAXP configuration file, `<java_home>/conf/jaxp.properties`
- The default values for JAXP properties; if FSP is true, then the default values will be set to process XML securely

Using the `RESOLVE` property of the `CatalogFeatures` class as an example, the following illustrates how these rules are applied:

- Properties specified through factory or processor APIs have the highest precedence. The following code effectively sets the `RESOLVE` property to `strict` regardless of settings in any other configuration sources.

  ```
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
  dbf.setAttribute(CatalogFeatures.Feature.RESOLVE.getPropertyName(),
  "strict");
  ```

- If the property is not set by the factory as in the previous example, a system property setting will be in effect. The following command-line example sets the `RESOLVE` property to `continue` for the application `myApp`:

  ```
  java -Djavax.xml.catalog.resolve=continue myApp
  ```

- If the property is not set by the factory or in a system property, the setting in a configuration file will take effect. The following entry sets the `RESOLVE` property to `continue`.

  ```
  javax.xml.catalog.resolve=continue
  ```

- If the value of the `RESOLVE` property is not set anywhere, it will be resolved to its default value, `strict`.

Security-Related Properties

The following sections describe the properties you can set to configure JAXP for secure XML processing:

There are two kinds of JAXP security-related properties:

- API-Defined Properties: These are constant field values that are part of the Java SE API whose values you set through a factory's or parser's `setAttribute(String, Object)` or `setProperty(String, Object)` method. Some of these fields have corresponding system properties you can set on the command line or in a JAXP configuration file:
  - The FEATURE_SECURE_PROCESSING Security Directive
  - External Access Properties

Copyright © 1993, 2026, Oracle and/or its affiliates.

- JDK-Specific Properties: These are system properties you set through JAXP parsers or factories, on the command line, or in a JAXP configuration file. Some of these aren't system properties, which means you can't set their values on the command line:
  - JAXP Properties for Processing Limits
  - JAXP Property for Extension Functions
  - JAXP Property for Third-Party Parsers
  - DTD Properties

API-Defined Properties

Topics

- The FEATURE_SECURE_PROCESSING Security Directive
- External Access Properties

The FEATURE_SECURE_PROCESSING Security Directive

The `FEATURE_SECURE_PROCESSING` (FSP) directive, defined by the JAXP API, signals the intent to enforce security restrictions during XML document processing. When set to `true` using a JAXP factory as follows, it instructs the JAXP processor (for example, a parser, validator, or transformer) to apply security controls based on the processor's security model:

```
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

When FSP is enabled, the JDK activates additional security measures using External Access Properties (EAPs) and JDK-specific properties. See External Access Properties (EAPs) and JDK-Specific Properties.

By default, the JDK turns FSP on for SAX, DOM, validation, and transformation factories. However, external connections (such as external entity resolution or resource access) are not disabled by default. To ensure external access is properly restricted, FSP needs to be explicitly set as shown previously.

External Access Properties

External access properties (EAPs), along with their corresponding System properties, enable you to regulate external connections.

EAPs enable you to specify the type of external connections that can or cannot be permitted. The property values are a list of protocols. The JAXP processors check if a given external connection is permitted by matching the protocol with those in the list. Processors will attempt to establish the connection if it is on the list, or reject it if not. Use these JAXP properties along with custom resolvers and the Catalog API (see Using Resolvers and Catalogs) to reduce the risk of external connections by rejecting and resolving them with local resources.

> **Note**
>
> External connections are allowed even when FSP is on by default. Explicitly turning on FSP through the API, for example, `factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true)`, is necessary to disable all external connections.

Copyright © 1993, 2026, Oracle and/or its affiliates.

The EAPs are defined in `javax.xml.XMLConstants` as follows:

- `javax.xml.XMLConstants.ACCESS_EXTERNAL_DTD`
- `javax.xml.XMLConstants.ACCESS_EXTERNAL_SCHEMA`
- `javax.xml.XMLConstants.ACCESS_EXTERNAL_STYLESHEET`

You can specify the values of these properties as a System property or in a JAXP configuration file.

Table 12-1 EAPs

Property	Description	Default Value
`javax.xml.accessExternalDTD`	Restricts access to external DTDs and external entity references to the protocols specified. See `javax.xml.XMLConstants.ACCESS_EXTERNAL_DTD`.	`all`, connection permitted to all protocols
`javax.xml.accessExternalSchema`	Restricts access to the protocols specified for external references set by the `schemaLocation` attribute, `import` element, and `include` element. See `javax.xml.XMLConstants.ACCESS_EXTERNAL_SCHEMA`.	`all`, connection permitted to all protocols
`javax.xml.accessExternalStylesheet`	Restricts access to the protocols specified for external references set by the `stylesheet` processing instruction, `document` function, and `import` and `include` elements. See `javax.xml.XMLConstants.ACCESS_EXTERNAL_STYLESHEET`	`all`, connection permitted to all protocols

Values of EAPs

All EAPs have values of the same format:

- **Value**: A list of protocols separated by comma. A protocol is the scheme portion of an URI, or in the case of the JAR protocol, `jar` plus the scheme portion separated by colon. A scheme is defined as:

  `scheme = alpha *( alpha | digit | "+" | "-" | "." )`
  where *`alpha`* = `a-z` and `A-Z`.

  The JAR protocol is defined as: `jar[:scheme]`
  Protocols are case-insensitive. Any whitespace characters as defined by `Character.isSpaceChar` in the value are ignored. Examples of protocols are `file`, `http`, and `jar:file`.

- **Default value**: The default value is implementation specific. For the JDK, the default value is `all`, which grants permissions to all protocols.

- **Granting all access**: The keyword `all` grants permission to all protocols. For example, specifying `javax.xml.accessExternalDTD=all` in the JAXP configuration file enables a

Copyright © 1993, 2026, Oracle and/or its affiliates.

system to work as before with no restrictions on accessing external DTDs and entity references.

- **Denying any access**: An empty string ("") means that no permission is granted to any protocol. For example, specifying `javax.xml.accessExternalDTD=""` in the JAXP configuration file instructs JAXP processors to deny any external connections.

When to Use External Access Properties

The XML processors, by default, attempt to connect and read external resources that are referenced in XML sources. Note that this may potentially expose applications and systems to risks posed by external connections. It's therefore recommended that applications consider limiting external connections with EAPs.

Internal applications and systems that handle only trusted XML documents may not need these EAPs. However, keep in mind that EAPs are specific to the XML processors and are at the top layer of the process, which means that the processors check these EAPs before any connections are made. They may therefore serve as an additional and more direct protection against external connection risks.

You can use EAPs along with custom resolvers and catalogs (see Using Resolvers and Catalogs) to effectively manage external connections and reduce risks.

Even in a trusted environment with trusted sources, it's recommended that you use both EAPs and resolvers to minimize dependencies on external sources.

JAXP Property Precedence and External Access Properties

EAPs have no effect on the relevant constructs that they attempt to restrict in the following situations:

- When there is a resolver and the source returned by the resolver is not null: This applies to entity resolvers that may be set on SAX and DOM parsers, XML resolvers on StAX parsers, `LSResourceResolver` on `SchemaFactory`, a `Validator` or `ValidatorHandler`, or `URIResolver` on a transformer.
- When a schema is created explicitly by calling the `newSchema` method from `SchemaFactory`.
- When external resources are not required: For example, the following features and properties are supported by the JDK and may be used to instruct the processor to not load the external DTD or resolve external entities:

```
http://apache.org/xml/features/disallow-doctype-decl true
http://apache.org/xml/features/nonvalidating/load-external-dtd false
http://xml.org/sax/features/external-general-entities false
http://xml.org/sax/features/external-parameter-entities false
```

JDK-Specific Properties

Topics

- JAXP Properties for Processing Limits
- JAXP Property for Extension Functions
- JAXP Property for Third-Party Parsers

Copyright © 1993, 2026, Oracle and/or its affiliates.

- DTD Properties

JAXP Properties for Processing Limits

XML processing can sometimes be a memory intensive operation. Applications, especially those that accept XML, XSD and XSL from untrusted sources, should take steps to guard against excessive memory consumption by using JAXP properties for processing limits.

Evaluate your application's requirements and operating environment to determine the acceptable processing limits for your system configurations and set these limits accordingly. For example, use size-related limits to prevent malformed XML sources from consuming large amounts of memory. Use the `jdk.xml.entityExpansionLimit` property to enable an application to control memory consumption under an acceptable level.

The JDK XML parsers observe processing limits by default. Both DOM and SAX parsers have FSP turned on by default and therefore turn on the limits. The StAX parser also observes processing limits by default even though it doesn't support FSP.

The following table describes the JAXP properties for processing limits supported in the JDK. The value for each of these processing limits is a positive integer. A value less than or equal to 0 indicates no limit. If the value is not an integer, a `NumericFormatException` is thrown. You can specify the values of these properties through its factory, as a System property, or in a JAXP configuration file.

See the table Implementation Specific Properties in the `java.xml` module summary for more information.

Table 12-2 JAXP Processing Limit Properties

Property Name	Description	Default Value	Supported Factories

`jdk.xml.elementAttributeLimit`	Limits the number of attributes an element can have.	200	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

`jdk.xml.entityExpansionLimit`	Limits the number of entity expansions.	2500	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

`jdk.xml.entityReplacementLimit`	Limits the total number of nodes in all entity references.	100000	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 12-2 (Cont.) JAXP Processing Limit Properties

Property Name	Description	Default Value	Supported Factories

`jdk.xml.maxElementDepth`	Limits the maximum element depth.	100	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

`jdk.xml.maxGeneralEntitySizeLimit`	Limits the maximum size of any general entities.	100000	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

`jdk.xml.maxOccurLimit`	Limits the number of content model nodes that may be created when building a grammar for a W3C XML Schema that contains `maxOccurs` attributes with values other than "unbounded".	5000	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

`jdk.xml.maxParameterEntitySizeLimit`	Limits the maximum size of any parameter entities, including the result of nesting multiple parameter entities.	15000	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

`jdk.xml.maxXMLNameLimit`	Limits the maximum size of XML names, including element name, attribute name and namespace prefix and URI.	1000	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

`jdk.xml.totalEntitySizeLimit`	Limits the total size of all entities that include general and parameter entities. The size is calculated as an aggregation of all entities.	100000	`DocumentBuilderFactory`
			`SAXParserFactory`
			`XMLInputFactory`
			`SchemaFactory`
			`TransformerFactory`

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 12-2 (Cont.) JAXP Processing Limit Properties

Property Name	Description	Default Value	Supported Factories
`jdk.xml.xpathExprGrpLimit`	Limits the number of groups an XPath expression can contain.	10	`TransformerFactory` `XPathFactory`
`jdk.xml.xpathExprOpLimit`	Limits the number of operators an XPath expression can contain.	100	`TransformerFactory` `XPathFactory`
`jdk.xml.xpathTotalOpLimit`	Limits the total number of XPath operators in an XSL Stylesheet.	100000	`TransformerFactory`

FEATURE_SECURE_PROCESSING Default Limitations

The following table describes which XML-related factory classes are disabled and which processing limits are set if `FEATURE_SECURE_PROCESSING` (FSP) is enabled.

Table 12-3 Default Limitations Set by FSP on XML-Related Factory Classes

XML-Related Factory Class	Enabled ?	Processing Limits

`DocumentBuilderFactory`	true	`jdk.xml.entityExpansionLimit = 2500`
		`jdk.xml.elementAttributeLimit = 200`
		`jdk.xml.maxOccurLimit = 5000`

`SAXParserFactory`	true	`jdk.xml.entityExpansionLimit = 2500`
		`jdk.xml.elementAttributeLimit = 200`
		`jdk.xml.maxOccurLimit = 5000`

`SchemaFactory`	true	`jdk.xml.maxOccurLimit = 5000`
`TransformerFactory`	false	Extension functions disabled
`XPathFactory`	false	Extension functions disabled

Limiting Entity Expansion

Limit the number of entity expansions by either setting the system property `jdk.xml.entityExpansionLimit` or the parser property `http://apache.org/xml/properties/entity-expansion-limit`. Both properties accept `java.lang.Integer` values. The parser throws a fatal error once it has reached the entity expansion limit. By default, `entityExpansionLimit` is set to 64,000.

The following command-line example sets the entity expansion limit to 10,000:

```
java -DentityExpansionLimit=10000 MyApp
```

The following code example sets the entity expansion limit to 10,000:

```
System.setProperty("jdk.xml.entityExpansionLimit","10000");
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The following code example sets the parser property `http://apache.org/xml/properties/entity-expansion-limit` to 10,000:

```
DocumentBuilderFactory dfactory = DocumentBuilderFactory.newInstance();
dfactory.setAttribute(
        "http://apache.org/xml/properties/entity-expansion-limit",
        new Integer("10000"));
    DocumentBuilder docBuilder = dbFactory.newDocumentBuilder();
```

Limiting the Number of Element Attributes

Limit the number of attributes in an element by either setting the system property `jdk.xml.elementAttributeLimit` or by setting the parser property `http://apache.org/xml/properties/elementAttributeLimit`. Both properties accept `Integer` values. By default, `jdk.xml.elementAttributeLimit` is set to 10,000. When the parser property `http://apache.org/xml/properties/elementAttributeLimit` is set, it overrides the system property. The parser throws a fatal error if the number of attributes in a element exceeds the limit.

The following command-line example sets the element attribute limit to 20:

```
java -Djdk.xml.elementAttributeLimit=20 MyApp
```

The following code example sets the element attribute limit to 20:

```
System.setProperty("jdk.xml.elementAttributeLimit","20");
```

The following code example sets the parser property `http://apache.org/xml/properties/entity-expansion-limit` to 20:

```
DocumentBuilderFactory dfactory = DocumentBuilderFactory.newInstance();
dfactory.setAttribute(
    "http://apache.org/xml/properties/elementAttributeLimit",
    new Integer(20));
DocumentBuilder docBuilder = dbFactory.newDocumentBuilder();
```

Limiting the Number of Nodes Created by Constructs That Contain maxOccurs

In constructs like `xsd:sequence`, the validating parser may use space (memory) proportional to the value of the `maxOccurs` occurrence indicator. This may cause the VM to run out of memory, or simply run for a very long time. To prevent potential attacks that exploit this behavior, enable secure processing on a factory as follows:

```
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, Boolean.TRUE);
```

Note that for `xsd:element` and `xsd:any`, the validating parser uses a constant amount of space, which is independent of the value of the `maxOccurs` occurrence indicator.

The default value of `jdk.xml.maxOccurLimit` is 5000. This system property limits the number of content model nodes that may be created when building a grammar for a W3C XML Schema that contains `maxOccurs` occurrence indicators with values other than "unbounded".

Copyright © 1993, 2026, Oracle and/or its affiliates.

When to Use Processing Limits

When determining which processing limits to apply and what values to use, at the system level, consider the amount of memory available for applications and whether XML, XSD, or XSL sources from untrusted sources are accepted and processed. At the application level, consider whether certain constructs such as DTDs are used.

Memory Setting and Limits

XML processing can be very memory intensive. The amount of memory that should be allowed to be consumed depends on the requirements of the applications in a specific environment. Processing of malformed XML data must be prevented from consuming excessive memory.

The default limits are generally set to allow legitimate XML inputs for most applications with memory usage allowed for a small hardware system, such as a PC. It is recommended that the limits are set to the smallest possible values, so that any malformed input can be caught before it consumes large amounts of memory.

The limits are correlated, but not entirely redundant. You should set appropriate values for all of the limits: usually the limits should be set to a much smaller value than the default.

For example, `ENTITY_EXPANSION_LIMIT` and `GENERAL_ENTITY_SIZE_LIMIT` can be set to prevent excessive entity references. But when the exact combination of the expansion and entity sizes are unknown, `TOTAL_ENTITY_SIZE_LIMIT` can serve as a overall control. Similarly, while `TOTAL_ENTITY_SIZE_LIMIT` controls the total size of a replacement text, if the text is a very large chunk of XML, `ENTITY_REPLACEMENT_LIMIT` sets a restriction on the total number of nodes that can appear in the text and prevents overloading the system.

Estimating the Limits Using the getEntityCountInfo Property

To help you analyze what values you should set for the limits, a special property called `http://www.oracle.com/xml/jaxp/properties/getEntityCountInfo` is available. The following code snippet, from Processing Limit Samples in *The Java Tutorials*, shows an example of using the property:

```
public static final String ORACLE_JAXP_PROPERTY_PREFIX =
    "http://www.oracle.com/xml/jaxp/properties/";
// ...
public static final String JDK_ENTITY_COUNT_INFO =
    ORACLE_JAXP_PROPERTY_PREFIX + "getEntityCountInfo";
// ...
parser.setProperty(JDK_ENTITY_COUNT_INFO, "yes");
```

When you run the processing limit sample with the DTD in W3C MathML 3.0, it prints the following table:

```
Sample: parsing W3C MathML


                      Property           Limit      Total size
Size                    Entity Name
        ENTITY_EXPANSION_LIMIT           64000            1417
0                           null
          MAX_OCCUR_NODE_LIMIT            5000               0
0                           null
       ELEMENT_ATTRIBUTE_LIMIT           10000               0
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
0                          null
       TOTAL_ENTITY_SIZE_LIMIT        50000000           59259
0                          null
     GENERAL_ENTITY_SIZE_LIMIT               0               0
10                          nvlt
   PARAMETER_ENTITY_SIZE_LIMIT         1000000               0
7304         %MultiScriptExpression
       MAX_ELEMENT_DEPTH_LIMIT             100               2
2                          null
                MAX_NAME_LIMIT            1000              13
13                          null
      ENTITY_REPLACEMENT_LIMIT         3000000               0
0                          null
             XPATH_GROUP_LIMIT              10               0
0                          null
                XPATH_OP_LIMIT             100               0
0                          null
           XPATH_TOTALOP_LIMIT           10000               0
0                          null
                          DTD               0               0
0                          null
           XERCES_DISALLOW_DTD               0               0
0                          null
             STAX_SUPPORT_DTD               1               0
0                          null
            JDKCATALOG_RESOLVE               0               0
0                          null
```

The "Limits" column suggests values you should set for the limits. In this example, it suggests 64000 for `ENTITY_EXPANSION_LIMIT`. However, there was a total of 1417 entity expansions.

If the DTD in W3C MathML 3.0 is the largest file that the application is expected to process, it is recommended that the limits be set to smaller numbers. For example, 2000 for `ENTITY_EXPANSION_LIMIT`, 100000 for `TOTAL_ENTITY_SIZE_LIMIT`, and 10000 for `PARAMETER_ENTITY_SIZE_LIMIT`.

JAXP Property for Extension Functions

The default value of the property for `jdk.xml.enableExtensionFunctions` is `false`, which disables XSLT and XPath external functions. External functions enable you to invoke your own or third-party code from a stylesheet.

If an application handles XML transformation with a stylesheet that uses extension functions, it may encounter a processing error, for example:

```
Use of the extension function <function name> is not allowed when extension
functions are disabled by the secure processing feature or the property
jdk.xml.enableExtensionFunctions. To enable extension functions, set
jdk.xml.enableExtensionFunctions to true.
```

You can set the property `jdk.xml.enableExtensionFunctions` to `true` as follows:

- Through the JAXP Transformation API (`javax.xml.transform` package):

  ```
  TransformerFactory tf = TransformerFactory.newInstance();
  tf.setFeature("jdk.xml.enableExtensionFunctions", true);
  ```


Copyright © 1993, 2026, Oracle and/or its affiliates.

- Through the XPath API (`javax.xml.xpath` package):

```
XPathFactory xf = XPathFactory.newInstance();
xf.setFeature("jdk.xml.enableExtensionFunctions", true);
```

- In the JAXP configuration file; see Setting JAXP Properties in a JAXP Configuration File
- On the command line:

```
java -Djdk.xml.enableExtensionFunctions=true myApp
```

JAXP Property for Third-Party Parsers

The JDK will always use its system-default parser even when there's a third-party parser on the classpath. To override the JDK system-default parser, set the `jdk.xml.overrideDefaultParser` property to `true` through `TransformerFactory`, `SchemaFactory`, or `XPathFactory`, as a System property, or in a JAXP configuration file.

Table 12-4 The jdk.xml.enableExtensionFunctions Property

Property	Description	Default Value	Supported Factories

`jdk.xml.overrideDefaultParser`	If `true`, then it enables the use of a third-party's parser implementation to override the system-default parser for the JDK's `Transformer`, `Validator`, and `XPath` implementations. If `false`, then it disables the use of third-party parser implementations.	`false`	`TransformerFactory`
			`SchemaFactory`
			`XPathFactory`

DTD Properties

If your applications don't require DTDs, then consider disabling DTD processing to safeguard against many common DTD-related attacks, including denial-of-service, XML external entity (XXE), and server-side request forgery (SSRF).

You can use the `jdk.xml.dtd.support` system property to disable DTD processing. It can have one of the following values:

- `allow`: The parser continues to process DTDs. This is the default value.
- `ignore`: The parser skips DTDs.
- `deny`: The parser rejects DTDs as an error. The parser reports the error in accordance with its specification.

Copyright © 1993, 2026, Oracle and/or its affiliates.

> **Note**
>
> Setting the value of the `jdk.xml.dtd.support` system property is the recommended way to disable DTD processing. This system property enables you to disable DTD processing for all parsers and processors. In addition, you can use this property in a JAXP configuration file.
> There are two other ways you can disable DTD processing, but they apply to specific parsers:
>
> - To disable DTD processing for SAX and DOM parsers, set the feature `http://apache.org/xml/features/disallow-doctype-decl` to `true` through a factory. The following code snippet disables DTDs for SAX parsers. A fatal error is thrown if the incoming XML document contains a DOCTYPE declaration.
>
>   ```
>   final static String DISALLOW_DTD =
>       "http://apache.org/xml/features/disallow-doctype-decl";
>   // ...
>   SAXParserFactory spf = SAXParserFactory.newInstance();
>   spf.setFeature(DISALLOW_DTD, true);
>   ```
>
> - To disable DTD processing for StAX parsers, set the property `SupportDTD` to `false` with the `XMLInputFactory.setProperty` method:
>
>   ```
>   XMLInputFactory xif = XMLInputFactory.newInstance();
>   xif.setProperty(XMLInputFactory.SUPPORT_DTD, Boolean.FALSE);
>   ```
>
> If you use any of these two ways to disable DTD processing, then the `jdk.xml.dtd.support` system property will have no effect.

Composite Processors

Composite processors such as the validator, transformer, and XPath processor use internally created parsers to read the source when it's not a `DOMSource` or `Document`. When `FEATURE_SECURE_PROCESSING` (FSP) is turned on through factories such as `SchemeFactory`, `TransformerFactory`, and `XPathFactory`, it's also on for any internal parser instance. This means the internal parser will behave the same as a parser created through `SAXParserFactory` or `DocumentBuilderFactory`.

For example, the following code creates an instance of `XPathFactory` with FSP:

```
XPathFactory xf = XPathFactory.newInstance();
xf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

This process ensures FSP is also on for any internal parser required by the XPath processor, such as when it's used to evaluate raw XML source. Consequently, any restrictions defined by the parser are applied. For instance, if the XML source contains external references not resolved by a catalog or resolver, it will be rejected as described in External Access Properties because FSP is turned on explicitly.

Copyright © 1993, 2026, Oracle and/or its affiliates.

Using Resolvers and Catalogs

You can register custom resolvers and catalogs on a JDK XML processor to intercept any references to external resources and resolve them with local ones. This feature eliminates the need to read and access external resources, thus helping to remove a source of potential risk.

Java XML Resolvers

The Java XML API supports various resolvers that you can register on JDK XML processors to resolve external resources. These resolvers includes entity resolvers for SAX and DOM parsers, XML resolvers for StAX parsers, `LSResourceResolver` for validation, and `URIResolver` for transformation.

Entity Resolvers for SAX and DOM

SAX defines an interface that DOM also supports, `org.xml.sax.EntityResolver`. It enables applications to step into the entity resolution process and perform entity resolution on their own terms. The following is the interface's definition:

```
package org.xml.sax;

public interface EntityResolver {
    public InputSource resolveEntity(String publicID, String systemID)
        throws SAXException;
}
```

You can then register an implementation of the interface on a SAX driver:

```
    EntityResolver resolver = ...;
    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setNamespaceAware(true);
    XMLReader reader = factory.newSAXParser().getXMLReader();
    reader.setEntityResolver(resolver);
```

Alternatively, you can register it on a DOM builder:

```
    DocumentBuilder builder =
        DocumentBuilderFactory.newInstance().newDocumentBuilder();
    docBuilder.setEntityResolver(resolver);
```

XMLResolver for StAX

StAX defines a `javax.xml.stream.XMLResolver` interface:

```
package javax.xml.stream;

public interface XMLResolver {
    public Object resolveEntity(
        String publicID, String systemID,
        String baseURI, String namespace)
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

```
        throws XMLStreamException;
}
```

You can register it on a StAX factory:

```
    XMLResolver resolver = ...;
    XMLInputFactory xif = XMLInputFactory.newInstance();
    xif.setProperty(XMLInputFactory.RESOLVER, resolver);
```

URIResolver for javax.xml.transform

The `javax.xml.transform` API supports custom resolution of external resources through the `URIResolver` interface:

```
package javax.xml.transform;

public interface URIResolver {
    public Source resolve(String href, String base)
        throws TransformerException;
}
```

You can register an implementation of `URIResolver` on a `Transformer` as follows:

```
    URIResolver resolver = ...;
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer t =
      tf.newTransformer(new StreamSource(
        new StringReader("xsl source")));
    t.setURIResolver(resolver);
```

LSResourceResolver for javax.xml.validation

The `javax.xml.validation` API supports Document Object Model Level 3 Load and Save (DOM LS) DOM through the `LSResourceResolver` interface:

```
package org.w3c.dom.ls;

public interface LSResourceResolver {
    public LSInput resolveResource(
        String type, String namespaceURI, String publicId,
        String systemId, String baseURI);
}
```

You can register an implementation of `LSResourceResolver` on a `SchemaFactory` as follows:

```
    SchemaFactory schemaFactory =
        SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
    LSResourceResolver resolver = ...;
    schemaFactory.setResourceResolver(resolver);
```


Copyright © 1993, 2026, Oracle and/or its affiliates.

The Catalog API

The XML Catalog API supports the Organization for the Advancement of Structured Information Standards (OASIS) XML Catalogs, OASIS Standard V1.1. This API is fully implemented by the JDK XML processors and easy to use. See XML Catalog API in *Java Platform, Standard Edition Core Libraries*.

Use the XML Catalog API to resolve external resources with the `CatalogResolver` interface and to enable catalogs on JDK XML processors.

Catalog Resolver

You can use a `CatalogResolver` as a custom resolver that substitutes external references with local resources configured as `Catalog` objects. You can register a `CatalogResolver` on factories or processors in place of `EntityResolver`, `XMLResolver`, `URIResolver` or `LSResourceResolver` as described in Java XML Resolvers. In the following code snippet, a `CatalogResolver` is registered as an `EntityResolver` on a `SAXParserFactory`:

```
    URI catalogUri = URI.create("file:///users/auser/catalog/catalog.xml")
    CatalogResolver cr =
        CatalogManager.catalogResolver(CatalogFeatures.defaults(),
catalogUri);
    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setNamespaceAware(true);
    XMLReader reader = factory.newSAXParser().getXMLReader();
    reader.setEntityResolver(cr);
```

Enable Catalogs on JDK XML Processors

The JDK XML processors implement the Catalog API as a native function. Therefore, there is no need to instantiate a `CatalogResolver` outside the processors. All you need to do is register the catalog files on the XML processors through the `setProperty` or `setAttribute` methods, through System properties, or in the JAXP configuration file. The XML processors then perform the mappings through the catalogs automatically. The following code snippet demonstrates how to register catalogs on StAX parsers through `XMLInputFactory`:

```
    String catalog = "file:///users/auser/catalog/catalog.xml";
    XMLInputFactory factory = XMLInputFactory.newInstance();
    factory.setProperty(CatalogFeatures.Feature.FILES.getPropertyName(),
catalog);
```

For more examples, see XML Catalog API in *Java Platform, Standard Edition Core Libraries*.

The Built-In Catalog

The JDK has a built-in catalog that hosts the following DTD and XSD files as defined by the Java platform and the World Wide Web Consortium (W3C):

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 12-5 DTD and XSD Files in the JDK Built-In Catalog

Source	DTD or XSD File or Files
java.util.prefs.Preferences	preferences.dtd
java.util.Properties	properties.dtd
XML Schema Part 1: Structures Second Edition, XML Schema Part 2: Datatypes Second Edition	XMLSchema.dtd, datatypes.dtd, XMLSchema.xsd, datatypes.xsd
XHTML 1.0 The Extensible HyperText Markup Language	xhtml1-frameset.dtd, xhtml1-strict.dtd xhtml1-transitional.dtd
XHTML 1.0 in XML Schema	xhtml1-frameset.xsd, xhtml1-strict.xsd xhtml1-transitional.xsd
XHTML 1.1 - Module-based XHTML - Second Edition	xhtml11.dtd
XHTML 1.1 XML Schema Definition	xhtml11.xsd
XML DTD for W3C specifications	xmlspec.dtd
The "xml:" Namespace	xml.xsd

The built-in catalog is loaded when the first JAXP processor factory is created.

The External Resource Resolution Process with the Built-In Catalog

The JDK creates a `CatalogResolver` with the built-in catalog when required. This `CatalogResolver` is the default external resource resolver.

XML processors may use resolvers (such as `EntityResolver`, `XMLResolver`, and `CatalogResolver`) to handle external references. In the absence of user-defined resolvers, the JDK XML processors fall back to the default `CatalogResolver` to attempt to find a resolution before making a connection to fetch the resources. The fallback also takes place if a user-defined resolver exists but allows the process to continue when unable to resolve the resource.

If the default `CatalogResolver` can't locate a resource, it may signal the XML processors to continue processing, ignore or skip the resource, or throw a `CatalogException`. The behavior is configured with the `jdk.xml.jdkcatalog.resolve` system property. It can have one of the following values:

- `continue`: Continue processing despite being unable to locate the resource. This is the default value.
- `ignore`: Skip the resource that can't be located.
- `strict`: Throw a `CatalogException` if the resource can't be located.

Handling Errors from JAXP Properties

It is recommended that applications catch `org.xml.sax.SAXNotRecognizedException` when setting JAXP properties so that the applications will work properly on older releases that don't support them.

Copyright © 1993, 2026, Oracle and/or its affiliates.

For example, the following method, `isNewPropertySupported`, from Processing Limit Samples in *The Java Tutorials*, detects if the sample is run with a version of the JDK that supports the `JDK_GENERAL_ENTITY_SIZE_LIMIT` property:

```
    public boolean isNewPropertySupported() {
        try {
            SAXParser parser = getSAXParser(false, false, false);
            parser.setProperty(JDK_GENERAL_ENTITY_SIZE_LIMIT, "10000");
        } catch (ParserConfigurationException ex) {
            fail(ex.getMessage());
        } catch (SAXException ex) {
            String err = ex.getMessage();
            if (err.indexOf("Property '" + JDK_GENERAL_ENTITY_SIZE_LIMIT +
                                                "' is not recognized.") > -1) {
                // expected before this patch
                debugPrint("New limit properties not supported. Samples not
run.");
                return false;
            }
        }
        return true;
    }
```

When input files contain constructs that cause an over-the-limit exception, applications may check the error code to determine the nature of the failure. The following error codes are defined for processing limits:

- `EntityExpansionLimit`: JAXP00010001
- `ElementAttributeLimit`: JAXP00010002
- `MaxEntitySizeLimit`: JAXP00010003
- `TotalEntitySizeLimit`: JAXP00010004
- `MaxXMLNameLimit`: JAXP00010005
- `maxElementDepth`: JAXP00010006
- `EntityReplacementLimit`: JAXP00010007

The error code has the following format:

`"JAXP" + components (two digits) + error category (two digits) + sequence number`

The code JAXP00010001, therefore, represents the JAXP base parser security limit `EntityExpansionLimit`.

If access to external resources is denied due to the restrictions set by EAPs, then an exception will be thrown with an error in the following format:

```
[type of construct]: Failed to read [type of construct]
    "[name of the external resource]", because "[type of restriction]"
    access is not allowed due to restriction set by the
    [property name] property.
```

For example, suppose the following:

Copyright © 1993, 2026, Oracle and/or its affiliates.

- The `ACCESS_EXTERNAL_DTD` JAXP property is set as follows:

```
parser.setProperty(
    "http://javax.xml.XMLConstants/property/accessExternalDTD", "file");
```

- Your application tries to fetch an external DTD with the HTTP protocol.
- The parser parsed an XML file that contains an external reference to `http://www.example.com/dtd/properties.dtd`.

The error message would look like the following:

```
External DTD: Failed to read external DTD
    "http://www.example.com/dtd/properties.dtd", because "http"
    access is not allowed due to restriction set by the
    accessExternalDTD property.
```

General Recommendations for JAXP Security

The following are general recommendations for configuring JAXP properties and features to help secure your applications and systems:

- Create and specify a user-defined JAXP configuration file with property settings tailored to your environment; see Setting JAXP Properties in a JAXP Configuration File. Afterward, adjust individual features and properties in accordance with specific requirements of the applications through factory settings or System properties.
- Disable DTDs with the `jdk.xml.dtd.support` system property if the application does not require or process DTD content. See DTD Properties.
- For processing limits, adjust them so that they are just large enough to accommodate the maximum amount your applications require.
- For EAPs, reduce or eliminate your applications' reliance on external resources, including the use of resolvers, then tighten the restrictions specified by EAPs.
- Set up a local catalog and enable the Catalog API on all XML processors to further reduce your applications' reliance on external resources.
- Reject any external references not resolved by a user-defined resolver or catalog by configuring the built-in catalog's `jdk.xml.jdkcatalog.resolve` system property. See The Built-In Catalog.

Appendix A: Glossary of Java API for XML Processing Terms and Definitions

Table 12-6 JAXP Glossary

Term	Definition
JAXP	Java API for XML Processing
Java SE XML API	APIs defined in the JAXP JSR and integrated into Java SE
Java XML API	Equivalent term for Java SE XML API

Copyright © 1993, 2026, Oracle and/or its affiliates.

Table 12-6 (Cont.) JAXP Glossary

Term	Definition
Java XML Features and Properties	XML-related features and properties defined by the Java SE specification
`java.xml`	The `java.xml` module
JDK XML	The JDK implementation of the Java XML API
JDK XML Parsers	The JDK implementation of the XML parsers
JDK XML Properties	The JDK Implementation-only properties
EAPs	External Access Properties
FSP	The FEATURE_SECURE_PROCESSING Security Directive

Appendix B: Java and JDK XML Features and Properties Naming Convention

Java and JDK XML features and properties are defined in the `javax.xml.XMLConstants` class. The features have a prefix `http://javax.xml.XMLConstants/feature`; the properties, `http://javax.xml.XMLConstants/property`. If there is a corresponding System property, its prefix is `javax.xml`.

The JDK XML properties are JDK implementation-only properties. The prefix of the properties is `http://www.oracle.com` for JDK 8 and earlier and `jdk.xml` for JDK 9 and later. The following table summarizes this naming convention:

Table 12-7 Java and JDK XML Features and Properties Naming Convention

Scope	API Property Prefix	System Property Prefix	Java SE and JDK Version
Java SE	`http://javax.xml.XMLConstants/feature` `http://javax.xml.XMLConstants/property`	`javax.xml`	Since 1.4
JDK	`http://www.oracle.com/xml/jaxp/properties`	`jdk.xml`	Since 7
JDK	`jdk.xml`	`jdk.xml`	Since 9

Copyright © 1993, 2026, Oracle and/or its affiliates.